

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/364302398>

Projeto e Analise de Algoritmos

Preprint · October 2022

CITATIONS

0

READS

2,829

1 author:



Alexandre L M Levada
Universidade Federal de São Carlos

130 PUBLICATIONS 389 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Encoded Blue Eyes Project (EBE) [View project](#)



Unsupervised metric learning [View project](#)



Departamento de Computação
Centro de Ciências Exatas e Tecnologia
Universidade Federal de São Carlos

Projeto e Análise de Algoritmos

Estratégias para análise, projeto de algoritmos e soluções aproximadas
para problemas NP-completos

Prof. Alexandre Luis Magalhães Levada
Email: alexandre.levada@ufscar.br

Sumário

Análise Assintótica.....	3
Dividir para conquistar.....	14
Árvores de recursão.....	15
O Problema da Multiplicação de Inteiros.....	17
O algoritmo de Karatsuba.....	19
O algoritmo Mergesort.....	22
O Problema da Multiplicação de Matrizes.....	24
O algoritmo de Strassen.....	27
O problema do par de pontos mais próximos.....	29
A Transformada de Fourier Discreta.....	34
O algoritmo Fast Fourier Transform (FFT).....	38
O Teorema Mestre.....	42
Programação Dinâmica.....	46
A série de Fibonacci.....	46
O problema da sequência de cédulas.....	48
O problema do robô coletor de moedas.....	50
O Problema do corte da haste (The rod cutting problem).....	52
Caminhos mínimos em grafos.....	56
O algoritmo Floyd-Warshall.....	62
Algoritmos Gulosos.....	66
Árvores geradoras mínimas.....	67
O algoritmo de Kruskal.....	71
Algoritmo de Prim.....	75
O código de Huffman.....	80
Fluxo em redes e o algoritmo de Ford-Fulkerson.....	87
Emparelhamentos em grafos bipartidos.....	99
Emparelhamentos estáveis e o algoritmo de Gale-Shapley.....	99
Emparelhamentos e o algoritmo Húngaro.....	106
O problema da alocação ótima (The optimal assignment problem).....	122
Problemas NP-Completos.....	131
Grafos Hamiltonianos.....	133
O problema do caixeiro-viajante.....	136
Coloração de vértices.....	152
Bibliografia.....	164
Sobre o autor.....	165

“Experiência não é o que acontece com um homem; é o que ele faz com o que lhe acontece”
(Aldous Huxley)

Análise Assintótica

Trata-se de uma ferramenta matemática para avaliar a performance de algoritmos em termos do tamanho da entrada n , contando o número de instruções a serem executadas.

Vantagem: não depende do hardware da máquina como o tempo de execução.

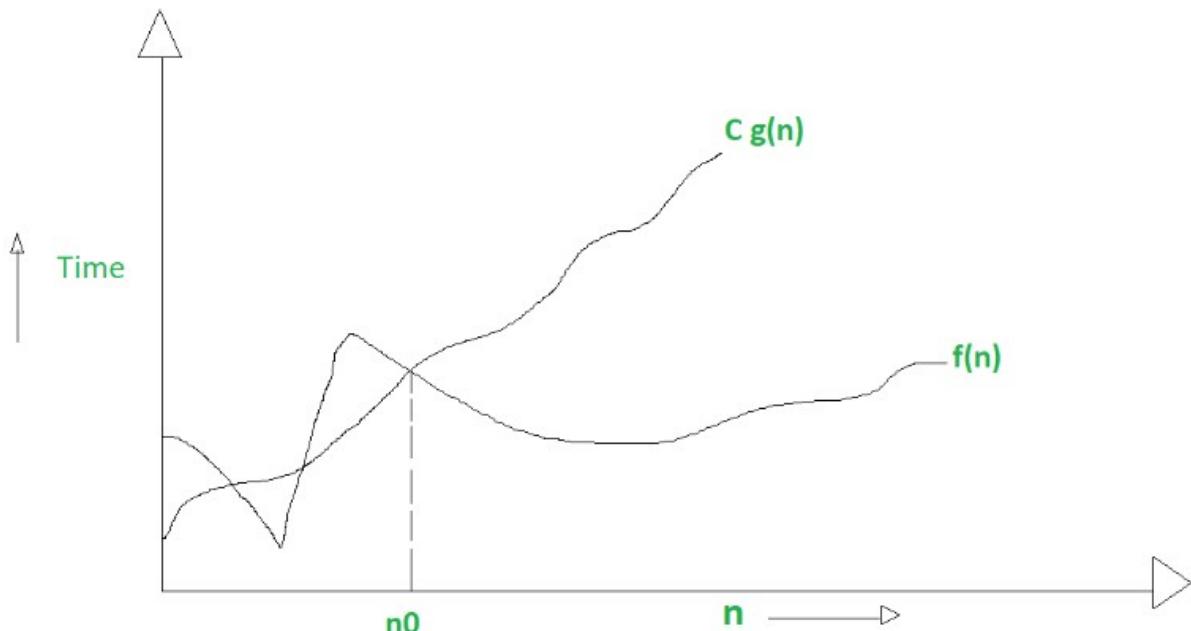
Na análise de algoritmos, o estudo do crescimento assintótico de funções deve ser empregado pois desejamos verificar o comportamento dos algoritmos para grandes valores de n (tamanho da entrada cresce arbitrariamente). Existem 3 notações básicas para análise assintótica de funções: Big-O, notação Ω e notação Θ . A seguir definiremos cada uma delas.

Notação Big-O

Representa o limite superior do tempo de execução de um algoritmo. Abordagem pessimista, especialmente adequada para estudar como algoritmo se comporta no pior cenário.

$$f(n) = O(g(n)) \text{ se há constantes positivas } c \text{ e } n_0 \text{ tais que } f(n) \leq c g(n) \quad \forall n \geq n_0$$

Em outras palavras, se para todo n maior que um limite arbitrário, $f(n)$ é limitada superiormente por $c g(n)$.



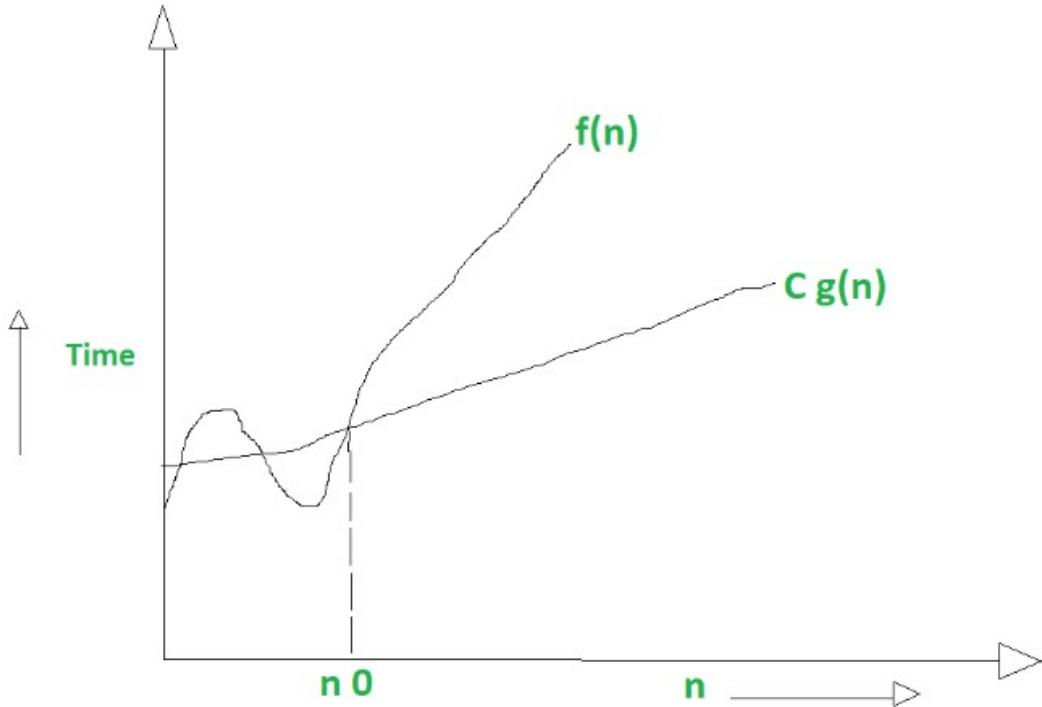
Um exemplo é o algoritmo de ordenação InsertionSort que é linear no melhor caso e quadrático no pior caso. Como ele nunca é pior que complexidade quadrática, podemos dizer que no geral esse algoritmo é $O(n^2)$.

Notação Ω

Representa o limite inferior do tempo de execução de um algoritmo. Abordagem otimista, empregada para estudar como um algoritmo se comporta no melhor cenário.

$$f(n) = \Omega(g(n)) \text{ se há constantes positivas } c \text{ e } n_0 \text{ tais que } f(n) \geq c g(n) \quad \forall n \geq n_0$$

Em outras palavras, se para todo n maior que um limite arbitrário, $f(n)$ é limitada inferiormente por $c g(n)$.



Por exemplo, o algoritmo InsertionSort possui complexidade sempre maior que linear, então podemos escrever que esse algoritmo é $\Omega(n)$.

Notação Θ

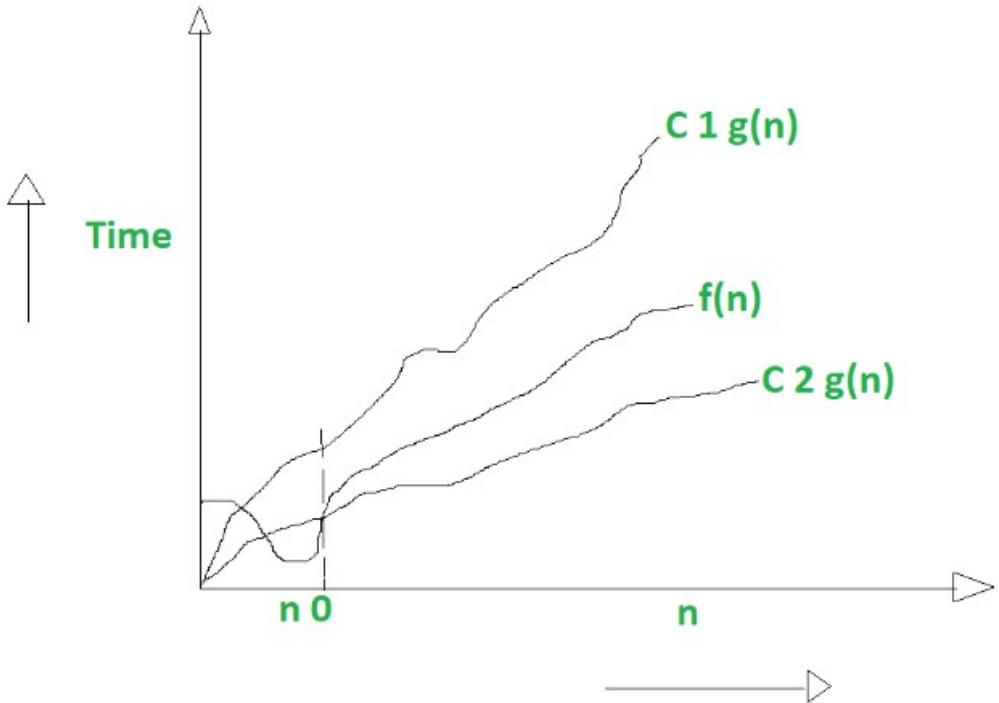
Essa notação limita a função acima e abaixo simultaneamente. Representa o limite superior e inferior do algoritmo em questão, especialmente adequada para estudar o comportamento de algoritmos no caso médio.

$$f(n) = \Theta(g(n)) \text{ se há constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tais que:}$$

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

para todo $n \geq n_0$

Na prática, é como se a função $f(n)$ ficasse presa entre $c_1 g(n)$ e $c_2 g(n)$ (limitada acima e abaixo). A figura a seguir ilustra esse comportamento.



Propriedades da notação assintótica

1. Se $f(n)=O(g(n))$ então $af(n)=O(g(n))$ para uma constante a

$$f(n)=2n^2+5 \text{ é } O(n^2)$$

$$7f(n)=14n^2+35 \text{ também é } O(n^2)$$

O mesmo vale para notações Ω e Θ .

2. Transitividade: se $f(n)=O(g(n))$ e $g(n)=O(h(n))$, então $f(n)=O(h(n))$.

$$\text{Seja } f(n)=n, g(n)=n^2 \text{ e } h(n)=n^3.$$

$$n=O(n^2) \text{ e } n^2=O(n^3), \text{ então } n=O(n^3)$$

O mesmo vale para notações Ω e Θ .

3. Se $f(n)=O(g(n))$, então $g(n)=\Omega(f(n))$

$$\text{Seja } f(n)=n \text{ e } g(n)=n^2$$

$$n=O(n^2) \text{ e } n^2=\Omega(n)$$

Esta propriedade vale apenas para Big-O e Ω .

4. Se $f(n)=O(g(n))$ e $f(n)=\Omega(g(n))$, então $f(n)=\Theta(g(n))$

5. Se $f(n)=O(g(n))$ e $d(n)=O(e(n))$, então:

$$f(n)+d(n)=O(\max\{g(n), e(n)\})$$

6. Se $f(n)=O(g(n))$ e $d(n)=O(e(n))$, então:

$$f(n)*d(n)=O(g(n)*e(n))$$

Se $f(n)=n$ e $d(n)=n^2$, então $f(n)*d(n)=O(n^3)$.

Obs: Matematicamente, dizemos que $O(n^2)$ representa um conjunto de funções. Alguns exemplos de funções que pertencem a esse conjunto são:

$$\begin{aligned} f(n) &= n^2 + n \\ f(n) &= n^2 + 1000n \\ f(n) &= 1000n^2 + 1000n \\ f(n) &= n \\ f(n) &= n^{1.999} \\ f(n) &= \frac{n^2}{\log n} \end{aligned}$$

O mesmo vale para $\Omega(n^2)$, que é um conjunto de funções. Exemplos de funções pertencentes a esse conjunto são:

$$\begin{aligned} f(n) &= n^2 - n \\ f(n) &= n^2 + n \\ f(n) &= 1000n^2 - 1000 \\ f(n) &= n^3 \\ f(n) &= n^{2.01} \\ f(n) &= 2^n \end{aligned}$$

A intuição é que podemos usar notações assintóticas como uma abstração de operadores relacionais:

$f(n)$ é $O(g(n))$ é aproximadamente o mesmo que $f(n) \leq g(n)$

$f(n)$ é $\Omega(g(n))$ é aproximadamente o mesmo que $f(n) \geq g(n)$

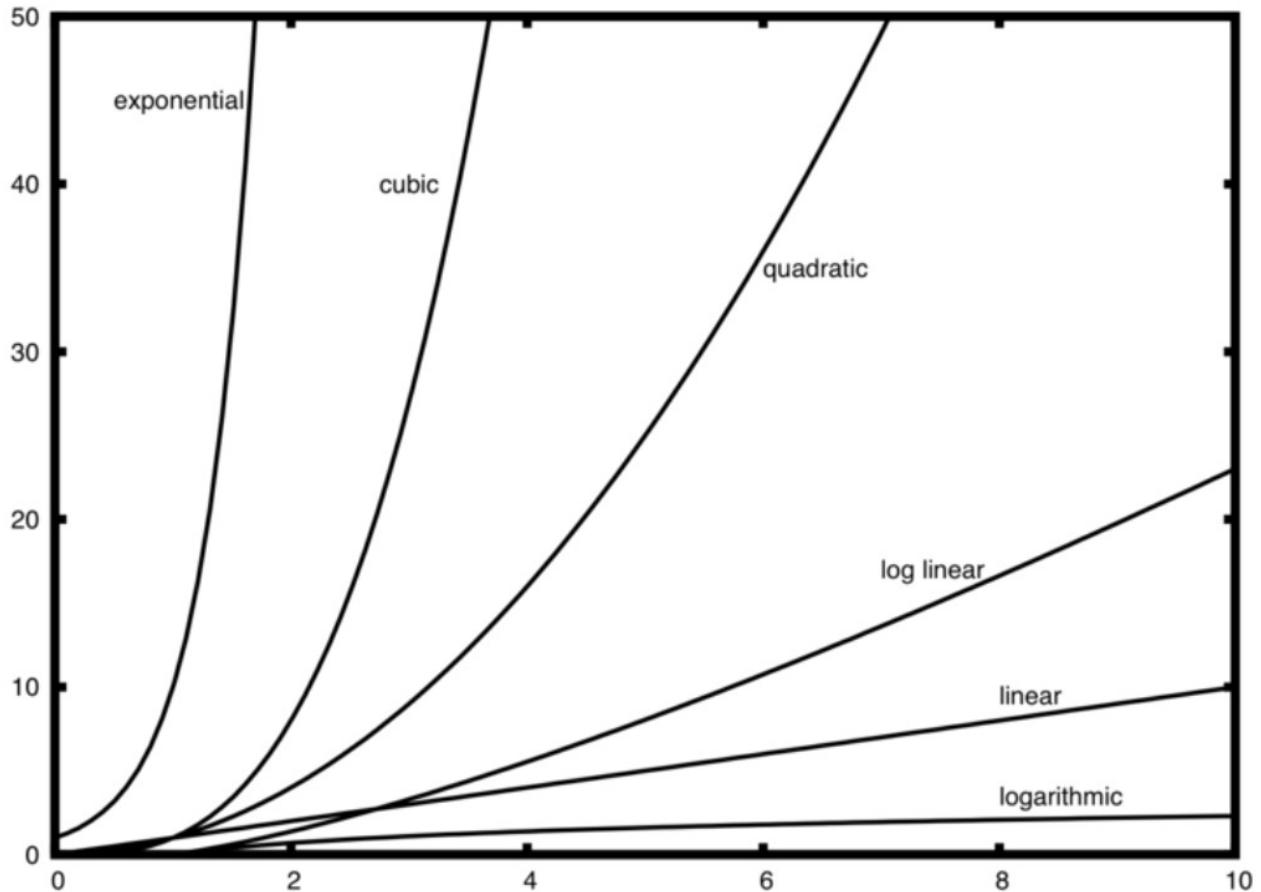
$f(n)$ é $\Theta(g(n))$ é aproximadamente o mesmo que $f(n) = g(n)$

Na prática, algumas regras que nos ajudam a simplificar funções omitindo termos dominados são:

1. Constantes multiplicadoras podem ser descartadas: $100n^2$ se torna n^2
2. A função n^a domina n^b , se $a > b$
3. Exponenciais dominam qualquer polinômio: 3^n domina n^8 , 3^n domina 2^n
4. Qualquer polinômio domina logaritmos: n domina $(\log n)^3$, n^2 domina $n \log n$

Classes de funções

Função	Classe
1	Constante
$\log n$	Logarítmica
n	Linear
$n \log n$	Log-Linear
n^k	Polinomial
k^n	Exponencial



Somatários

Séries e somatórios aparecem com frequência em diversos problemas da matemática e da computação. Na análise de algoritmos é bastante comum termos que resolver somatórios.

Para iniciar com um exemplo simples, suponha que desejamos somar todas as potências de 2, iniciando em 2^1 e terminando em 2^{10} . É possível expressar esse somatório como:

$$\sum_{k=1}^{10} 2^k = 2 + 2^2 + 2^3 + 2^4 + \dots + 2^{10}$$

Somas telescópicas: considere uma sequencia de números reais $x_1, x_2, x_3, \dots, x_n, x_{n+1}$. Então, a identidade a seguir é válida:

$$\sum_{k=1}^n (x_{k+1} - x_k) = x_{n+1} - x_1$$

ou seja, o valor do somatório das diferenças é igual a diferença entre o último elemento e o primeiro

Prova:

1. Pela propriedade associativa, temos:

$$S = \sum_{k=1}^n (x_{k+1} - x_k) = \sum_{k=1}^n x_{k+1} - \sum_{k=1}^n x_k$$

2. Por substituição de variáveis, temos:

$$S = \sum_{i=2}^{n+1} x_i - \sum_{k=1}^n x_k$$

3. Removendo o último termo do primeiro somatório e o primeiro termo do segundo, temos:

$$S = \sum_{i=2}^n x_i + x_{n+1} - x_1 - \sum_{k=2}^n x_k = x_{n+1} - x_1$$

A prova está concluída.

Analise de algoritmos

Objetivo: contar o número de atribuições no algoritmo A.

Hipóteses:

1. Cada comando de atribuição é $O(1)$
2. A função $T(n)$ deve contar o número de atribuições total a serem executadas quando entrada possui tamanho n

Considere o exemplo a seguir, com duas estruturas de repetição.

```
def FuncA(n):  
    c = 0  
    for i in range(n):  
        c = c + 1  
    for j in range(2n):  
        c = 2*c  
    return c
```

Note que temos uma atribuição inicial (1) e logo dois loops: o primeiro com n iterações e o segundo com $2n$ iterações. Assim, temos $T(n) = 3n + 1$, o que resulta em uma complexidade $O(n)$. Neste caso podemos escrever que: $T(n) = O(n)$ ou $T(n) = \Omega(n)$ ou $T(n) = \Theta(n)$.

Na verdade, podemos dizer que essa função é $\Omega(1)$, mas busca-se em geral os limites mais justos possíveis. De fato, qualquer função é $\Omega(1)$, mas isso não nos diz absolutamente nada sobre o comportamento assintótico dela.

Considere o algoritmo a seguir:

```
def FuncB(n):  
    c = 0  
    for i in range(n):  
        for j in range(n):  
            c = c + 1  
    return c
```

Nesse caso, o loop interno tem n operações. Como o loop externo é executado n vezes, e temos uma inicialização, o total de operações é $T(n)=n^2+1$, o que resulta em $O(n^2)$, $\Omega(n^2)$ ou $\Theta(n^2)$.

Considere esse código em que o loop interno executa um número variável de vezes.

```
def FuncC(n):
    c = 0
    for i in range(n) :
        for j in range(i+1) :
            c += 1
    return count
```

Note que quando $i = 0$, o loop interno executa uma vez, quando $n = 1$, o loop interno executa duas vezes, quando $n = 2$, o loop interno executa 3 vezes, e assim sucessivamente. Assim, o número de vezes que a variável c é incrementada é igual a: $1 + 2 + 3 + 4 + \dots + n$

Devemos resolver esse somatório para calcular a complexidade dessa função.

$$\sum_{k=1}^n k$$

Primeiramente, note que

$$(k+1)^2 = k^2 + 2k + 1$$

o que implica em

$$(k+1)^2 - k^2 = 2k + 1$$

Assim, $\sum_{k=1}^n [(k+1)^2 - k^2] = \sum_{k=1}^n [2k + 1]$. Porém, o lado esquerdo é uma soma telescópica e temos:

$$\sum_{k=1}^n [(k+1)^2 - k^2] = (n+1)^2 - 1$$

Dessa forma, podemos escrever:

$$(n+1)^2 - 1 = \sum_{k=1}^n [2k + 1]$$

Aplicando a propriedade associativa, temos:

$$(n+1)^2 - 1 = 2 \sum_{k=1}^n k + \sum_{k=1}^n 1$$

o que nos leva a:

$$2 \sum_{k=1}^n k = n^2 + 2n + 1 - 1 - n = n^2 + n$$

Finalmente, colocando n em evidência e dividindo por 2, finalmente temos:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Portanto, T(n) é igual a:

$$T(n) = \frac{1}{2}(n^2 + n) + 1$$

o que resulta em $O(n^2)$, $\Omega(n^2)$ ou $\Theta(n^2)$.

Considere a função em Python a seguir.

```
def FuncD(n):
    c = 0
    i = n
    while i > 1:
        c = c + 1
        i = i // 2      # divisão inteira
    return c
```

Essa função calcula quantas vezes o número pode ser dividido por 2. Por exemplo, considere a entrada n = 16. Em cada iteração esse valor será dividido por 2, até que atinja o zero.

$16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

A variável c termina a função valendo 4, pois $2^4 = 16$.

Se n = 25, temos:

$25 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 1$

A variável c termina a função valendo 4, pois $2^4 < 25 < 2^5$

Se n = 40, temos:

$40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 2 \rightarrow 1$

A variável c termina a função valendo 5, pois $2^5 < 40 < 2^6$

Portanto, o número de iterações do loop é $\log_2 n$. Dentro do loop existem duas instruções, portanto neste caso teremos:

$T(n) = 2 + 2\lfloor \log_2 n \rfloor$, onde a função piso(x) retorna o maior inteiro menor que x.

o que resulta em $O(\log_2 n)$, $\Omega(\log_2 n)$ ou $\Theta(\log_2 n)$.

Considere a função em Python a seguir.

```
def FuncE(n):
    c = 0
    for i in range(n):
        c = c + FuncD(n)
    return c
```

Note que, como a função FuncD(n) tem complexidade logarítmica, e o loop tem n iterações, temos que a complexidade da função em questão é $O(n \log_2 n)$, $\Omega(n \log_2 n)$ ou $\Theta(n \log_2 n)$.

Exercício: Mostre que, se c é um número real positivo, então:

$$g(n) = \sum_{k=0}^n c^k$$

é uma função:

- a) $\Theta(1)$, se $c < 1$.
- b) $\Theta(n)$, se $c = 1$
- c) $\Theta(c^n)$, se $c > 1$

Exercício: Mostre que para quaisquer constantes reais $a, b > 0$ $(n+a)^b$ é $\Theta(n^b)$.

Fatoriais

Sabemos que o fatorial de um número n é definido por:

$$n! = \begin{cases} 1, & n=0 \\ n(n-1)!, & n>0 \end{cases}$$

ou seja, $n! = \prod_{i=1}^n i$

Note que

$$n! = n(n-1)(n-2)\dots 1 \leq n \times n \times n \times \dots \times n = n^n$$

o que implica em $n! \leq n^n$ e portanto a conclusão de que $n!$ é $O(n^n)$.

Aproximação de Stirling

Pode-se mostrar que o fatorial de um inteiro pode ser computado por:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Então, para $n > 10$, $\Theta\left(\frac{1}{n}\right)$ torna-se desprezível, o que nos leva a:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Dessa forma, note que:

$$\log n! = \log(2\pi n)^{1/2} + n \log\left(\frac{n}{e}\right) = \frac{1}{2} \log 2\pi + \frac{1}{2} \log n + n \log n + n \log e$$

Como o primeiro e o último termos da soma são constantes e $n \log n$ domina $\log n$, chegamos a conclusão de que $\log n!$ é $O(n \log n)$.

Limites para comparação assintótica

Podemos comparar o crescimento assintótico de duas funções calculando o limite da razão entre elas. Apenas 3 casos podem ocorrer:

1. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$: significa que $f(n)$ cresce menos que $g(n)$, ou seja, $f(n) = O(g(n))$
2. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$: significa que $f(n)$ cresce proporcional a $g(n)$, ou seja, $f(n) = \Theta(g(n))$
3. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$: significa que $f(n)$ cresce mais rápido que $g(n)$, ou seja, $f(n) = \Omega(g(n))$

Regra de L'Hôpital

Pode-se mostrar que:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

onde $f'(n)$ denota a derivada da função f . Esse resultado é útil quando no cálculo de limites ocorrem indeterminações do tipo $0/0$, ∞/∞ , etc.

Compare o crescimento assintótico das funções $f(n) = \frac{1}{2}n(n-1)$ e $g(n) = n^2$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}$$

Portanto, temos que $f(n) = \Theta(g(n))$.

Compare o crescimento assintótico das funções $f(n) = \log_2 n$ e $g(n) = \sqrt{n}$.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'}$$

Para calcular a derivada de $f(n)$, devemos mudar a base para e (\ln):

$$\log_2 n = \frac{\ln n}{\ln 2}$$

Derivando em relação a n, temos:

$$\log_2 n = \left(\frac{\ln n}{\ln 2} \right)' = \frac{1}{\ln 2} \frac{1}{n}$$

Para o denominador temos:

$$(\sqrt{n})' = (n^{1/2})' = \frac{1}{2\sqrt{n}}$$

Assim, podemos escrever:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 2 \frac{1}{\ln 2} \frac{\lim_{n \rightarrow \infty} \sqrt{n}}{n} = 2 \frac{1}{\ln 2} \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$

Portanto, temos que $f(n) = \Omega(g(n))$, ou seja $f(n)$ cresce menos que $g(n)$. Isso significa que \sqrt{n} cresce mais rápido que $\log n$, ou seja, $g(n) = O(f(n))$.

Exercício: Compare as ordens de crescimento das funções $f(n) = n!$ e $g(n) = 2^n$.

"Enquanto o conhecimento custa o seu tempo no presente, a ignorância custa todo seu futuro."
-- Autor desconhecido

Dividir para conquistar

Ao projetar algoritmos, podemos empregar diversas estratégias para torná-los melhores. A estratégia dividir para conquistar (divide and conquer) é uma delas.

Trata-se de uma estratégia para projetar algoritmos assintoticamente eficientes.

Define um ferramental matemático para a resolução de recorrências.

Utiliza-se a recursão para simplificar um problema em instâncias menores e mais simples.

Estratégia dividir para conquistar

Para aplicar a estratégia dividir para conquistar devemos seguir alguns passos:

1. Caso base: se o problema é pequeno o suficiente, resolva-o diretamente (para a recursão).
2. Caso recursivo: caso contrário, realize 3 passos

- a) Divida o problema em subproblemas que são instâncias menores do problema original.
- b) Conquiste os subproblemas, resolvendo-os recursivamente.
- c) Combine as soluções dos subproblemas para formar a solução do problema original.

Objeto de estudo: recorrências que descrevem o número de instruções de algoritmos que utilizam a estratégia dividir para conquistar.

Um exemplo clássico que utiliza essa estratégia é o MergeSort. Trata-se de um algoritmo recursivo que divide uma lista continuamente pela metade. Se a lista estiver vazia ou tiver um único elemento, ela está ordenada por definição (o caso base). Se a lista tiver mais de um elemento, dividimos a lista e invocamos recursivamente um Mergesort em ambas as metades. Assim que as metades estiverem ordenadas, a operação fundamental, chamada de **intercalação**, é realizada. Intercalar é o processo de pegar duas listas menores ordenadas e combiná-las de modo a formar uma lista nova, única e ordenada.

Os três passos úteis dos algoritmos de dividir para conquistar que se aplicam ao MergeSort são:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante $O(1)$;
2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho $n/2$, o que contribui com $T(n/2) + T(n/2)$ para o tempo de execução;
3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo $O(n)$;

Assim, podemos escrever a relação de recorrência como:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Saber resolver esse tipo de recorrência é importante para podermos comparar a eficiência de diferentes algoritmos: Por exemplo, suponha 2 algoritmos A e B com as seguintes recorrências:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n^2)$$

Pergunta-se: qual deles é o mais eficiente assintoticamente? Veremos a seguir o método da árvore de recursão e como podemos utilizá-lo para responder perguntas como esta.

Árvores de recursão

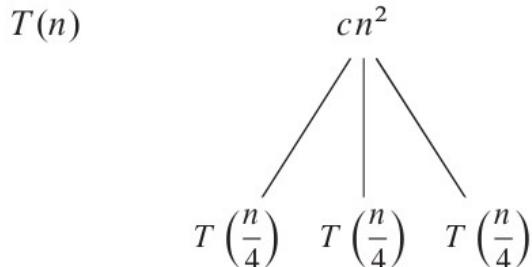
Decompor problema original (raiz da árvore) em uma sequência de subproblemas menores. Somando os custos dentro de cada nível da árvore temos o custo de uma recursão. Ao somar todos os níveis, chega-se ao custo total do algoritmo. Vejamos um exemplo ilustrativo. Seja a seguinte recorrência:

$$T(n) = 3T\left(\frac{n}{4}\right) + O(n^2)$$

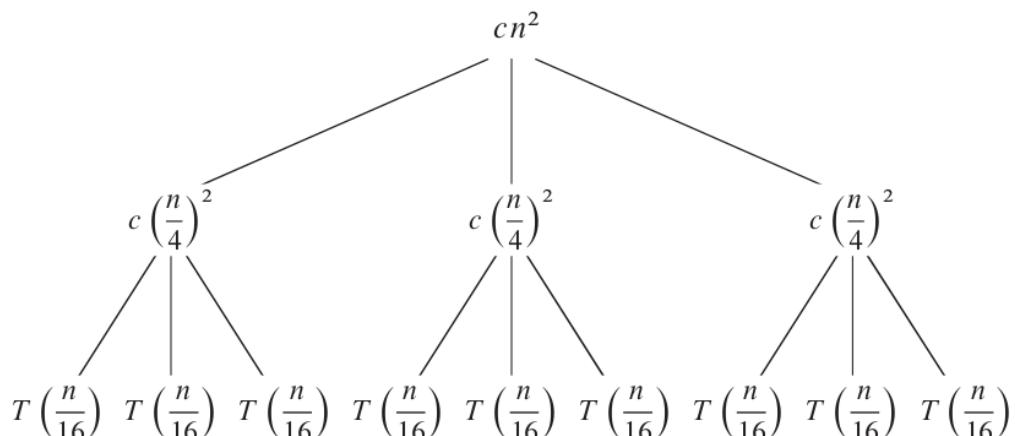
Essa recorrência nos diz que cada problema é dividido em 3 subproblemas de tamanho $n/4$ e o custo computacional para combinar as soluções dos subproblemas é quadrática. Se temos uma complexidade $O(n^2)$, significa que ela é limitada superiormente por cn^2 . Assim, a recorrência fica:

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$

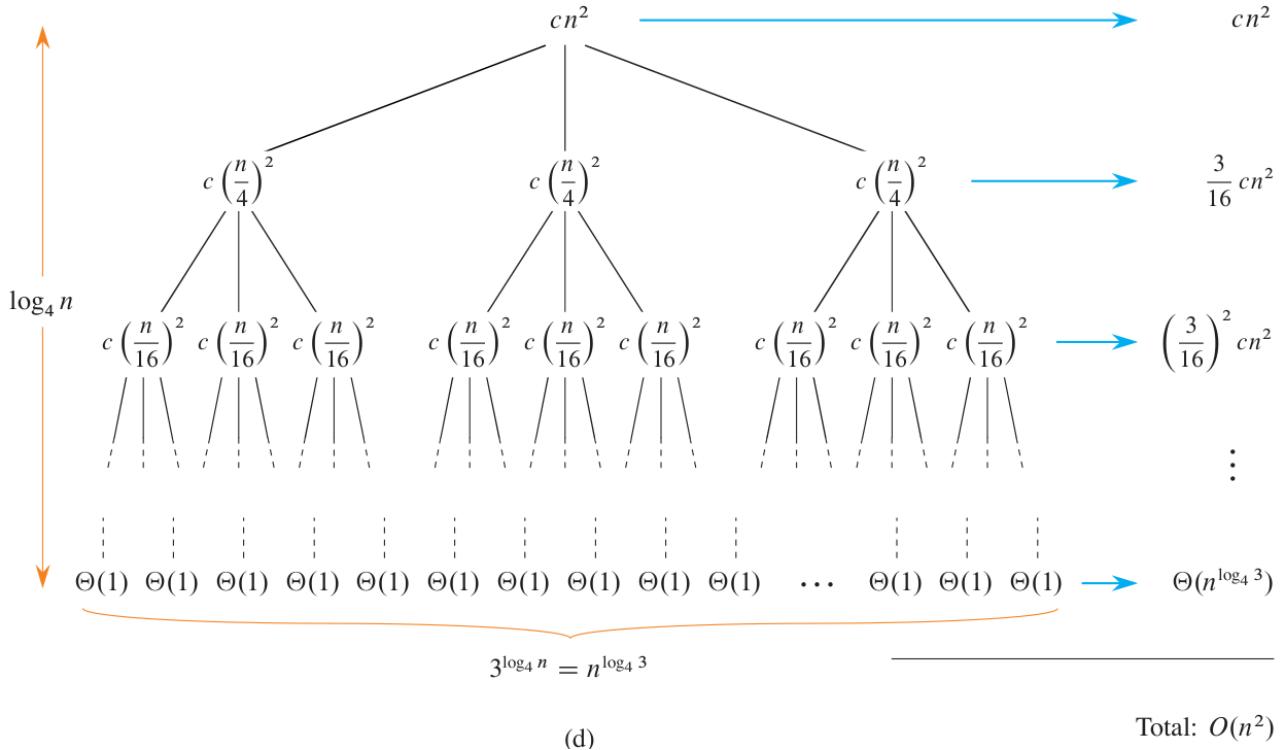
Podemos crescer uma árvore de recursão com raiz $T(n)$. Expandindo o primeiro nível da árvore, temos a seguinte construção:



A expansão do segundo nível, gera:



Continuando com o processo, chega-se a seguinte árvore de recursão:



O que precisamos determinar é a altura da árvore e a quantidade de nós folha no último nível. Note que no último nível, o custo é constante, ou seja, $O(1)$. Para que isso ocorra, devemos ter:

$$\frac{n}{4^k} = 1$$

o que implica que a altura da árvore é $k = \log_4 n$. Note que os números de nós em cada nível formam a seguinte sequência: 1, 3, 9, 27, 81, ... Assim, o número de nós na última camada será:

$$F = 3^k = 3^{\log_4 n}$$

Utilizaremos aqui uma importante propriedade dos logaritmos. Note que:

$$a^{\log_b c} = (b^{\log_b a})^{\log_b c} = b^{\log_b a \log_b c} = (b^{\log_b c})^{\log_b a} = c^{\log_b a}$$

Portanto, temos que o número total de nós folhas é $3^{\log_4 n} = n^{\log_4 3}$.

Somando os custos obtidos em cada nível da árvore de recursão, chega-se a:

$$T(n) = cn^2 + \left(\frac{3}{16}\right)cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \dots + \left(\frac{3}{16}\right)^{\log_4 n} cn^2 = \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2$$

Note que a expressão define a soma de uma PG de razão menor que 1. Fazendo o número de termos da PG ser igual a infinito, temos um limite superior para $T(n)$:

$$T(n) \leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2$$

A soma de uma PG infinita é dada por:

$$S_{\infty} = \frac{1}{1-q}$$

onde $q < 1$ é a razão da PG.

Sendo assim, a soma infinita é dada por:

$$S_{\infty} = \frac{1}{1-3/16} = \frac{16}{13}$$

o que nos leva a :

$$T(n) \leq \frac{16}{13} cn^2$$

o que é $O(n^2)$.

O Problema da Multiplicação de Inteiros

Nesta seção, iremos projetar e analisar um algoritmo mais eficiente para a multiplicação de números inteiros. Primeiramente, lembre-se que a multiplicação de 2 inteiros de n dígitos requer n^2 multiplicações e 2 adições:

$$\begin{array}{r} 384 \\ \times 156 \\ \hline 2304 \\ 1920 + \\ 384 + \\ \hline 59904 \end{array} \quad \begin{array}{l} \text{3 dígitos} \\ \text{3 dígitos} \\ \text{São 9 multiplicações + 2 adições = } O(n^2) \end{array}$$

A pergunta é: podemos fazer melhor que isso? Iremos tentar.

Sem perda de generalidade, é usual assumir que $n=2^m$ (potência de 2) para simplificar as coisas. Assim, sempre podemos particionar a e b em suas metades superiores e inferiores.

Por hora, vamos considerar os números a seguir:

$$\begin{aligned} a &= 123456 \\ b &= 654321 \end{aligned}$$

Note que podemos escrever:

$$\begin{aligned} a &= 123000 + 456 \\ b &= 654000 + 321 \end{aligned}$$

Logo, o produto entre a e b pode ser expresso como:

$$a \times b = (123 \times 10^3 + 456)(654 \times 10^3 + 321)$$

Aplicando a propriedade distributiva:

$$a \times b = 123 \times 456 \times 10^6 + (123 \times 321 + 456 \times 654) \times 10^3 + 456 \times 321$$

Usando essa notação, podemos escrever:

$$\begin{aligned} a &= a_1 10^{n/2} + a_2 \\ b &= b_1 10^{n/2} + b_2 \end{aligned}$$

onde a_1 e a_2 são as partes superiores e inferiores de a e b_1 e b_2 são as partes superiores e inferiores de b. A multiplicação é dada:

$$a \times b = A \times 10^n + (B+C) \times 10^{n/2} + D$$

onde

$$\begin{aligned} A &= a_1 \times b_1 \\ B &= a_2 \times b_1 \\ C &= a_2 \times b_1 \\ D &= a_2 \times b_2 \end{aligned}$$

A função recursiva a seguir utiliza a estratégia dividir para conquistar para implementar uma solução alternativa para a multiplicação de inteiros.

```
Multiply(a, b) {
    if len(a) <= 1
        return a*b
    Particione a e b em
        a=a_1 10^{n/2}+a_2      (essa partição requer um único loop) - O(n)
        b=b_1 10^{n/2}+b_2
    A = Multiply(a_1, b_1)
    B = Multiply(a_1, b_2)
    C = Multiply(a_2, b_1)
    D = Multiply(a_2, b_2)

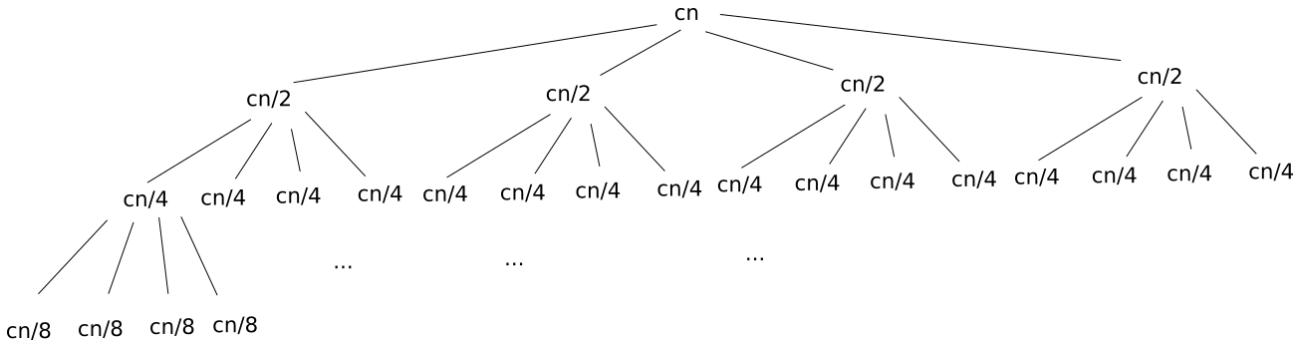
    P=A\times 10^n+(B+C)\times 10^{n/2}+D

    return P
}
```

Note que um problema de tamanho n é dividido em 4 problemas de tamanho n/2 mais um particionamento que tem complexidade O(n). Assim, a recorrência fica:

$$T(n) = 4 T\left(\frac{n}{2}\right) + O(n)$$

Podemos aplicar o método da árvore de recursão para gerar a seguinte estrutura.



Note que temos o seguinte padrão:

Nível 0: cn

Nível 1: $4 cn/2 = 2 cn$

Nível 2: $16cn/4 = 4 cn$

Nível 3: $64cn/8 = 8 cn$

...

$$\text{Nível } k: 4^k \frac{cn}{2^k} = 2^k cn$$

O último nível da árvore ocorre quando $\frac{n}{2^k} = 1$, ou seja, a altura da árvore é $k = \log_2 n$.

Portanto, temos:

$$T(n) = \sum_{i=1}^{\log_2 n} 2^i cn = cn \sum_{i=1}^{\log_2 n} 2^i$$

Lembre-se que $2^{i+1} = 2 \cdot 2^i = 2^i + 2^i$, o que nos leva a $2^i = 2^{i+1} - 2^i$. Dessa forma, temos:

$$T(n) = cn \sum_{i=1}^{\log_2 n} 2^{i+1} - 2^i = cn(2^{\log_2 n + 1} - 2^0) = cn(2^{2 \log_2 n} - 1) = cn(2n - 1) = 2cn^2 - cn$$

o que é $O(n^2)$. Conclusão: esse algoritmo não é mais eficiente que o usual!
Podemos melhorar diminuindo o número de subproblemas.

O algoritmo de Karatsuba

Note que no método anterior, o produto é dada por:

$$P = A \times 10^n + (B+C) \times 10^{n/2} + D$$

É possível reescrever a expressão $(B + C)$ como:

$$B+C = a_2 b_1 + a_1 b_2 = a_1 b_1 + a_2 b_1 + a_1 b_2 + a_2 b_2 - a_1 b_1 - a_2 b_2 = (a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2$$

ou seja, utilizamos apenas 3 produtos e não mais 4 como antes.

A seguir apresentamos o algoritmo de Karatsuba para multiplicar inteiros.

```

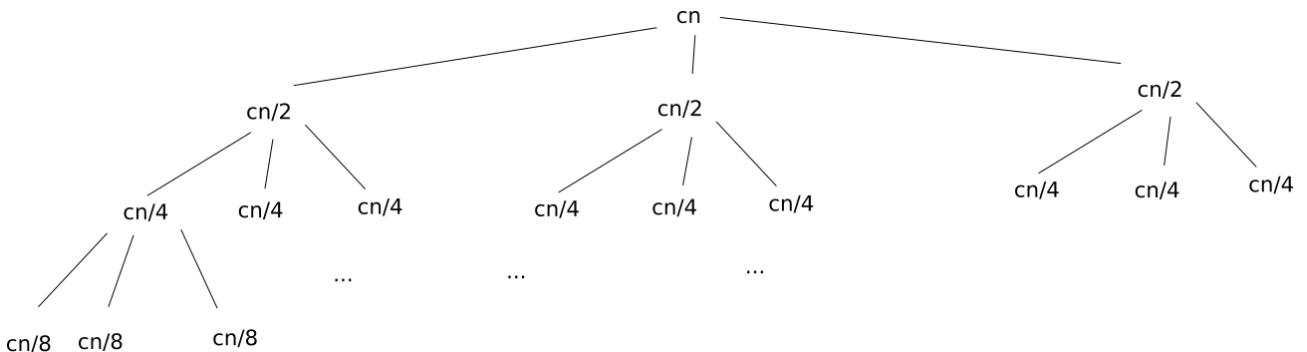
Multiply_K(a, b) {
    if len(a) <= 1
        return a*b
    Particione a e b em
        a=a110n/2+a2      (essa partição requer um único loop) - O(n)
        b=b110n/2+b2
    A = Multiply_K(a1, b1)
    B = Multiply_K(a2, b2)
    C = Multiply_K(a1 + a2, b1 + b2)
    P=A×10n+(C-A-B)×10n/2+D
    return P
}

```

Sendo assim, qual seria a complexidade computacional agora? Note que a recorrência pode ser expressa como:

$$T(n)=3T\left(\frac{n}{2}\right)+O(n)$$

A árvore de recorrência pode ser construída como:



Note que temos o seguinte padrão:

- Nível 0: cn
- Nível 1: 3/2 cn
- Nível 2: 3 (3/4) cn = 9/4 cn
- Nível 3: 9 (3/8) cn = 27/8 cn
- ...

$$\text{Nível } k: \quad 3^k \frac{cn}{2^k} = \left(\frac{3}{2}\right)^k cn$$

O último nível da árvore ocorre quando $\frac{n}{2^k}=1$, ou seja, a altura da árvore é $k=\log_2 n$.

Portanto, temos:

$$T(n)=\sum_{i=1}^{\log_2 n} \left(\frac{3}{2}\right)^i cn = cn \sum_{i=1}^{\log_2 n} \left(\frac{3}{2}\right)^i$$

Note que o somatório define a soma de uma P.G. em que o primeiro elemento é 1 e a razão $q = 3/2$.

$$\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \left(\frac{3}{2}\right)^2 + \left(\frac{3}{2}\right)^3 + \dots + \left(\frac{3}{2}\right)^{\log_2 n}$$

O número de termos n é igual a $\log_2 n + 1$. Lembrando que a soma da P.G. nada mais é que:

$$S_n = a_1 + a_2 + a_3 + \dots + a_n$$

Mas na P.G. sabemos que $a_n = a_1 q^{n-1}$, o que nos leva a:

$$S_n = a_1 + a_1 q + a_1 q^2 + \dots + a_1 q^{n-1} \quad (\text{I})$$

Multiplicando ambos os lados por q , temos:

$$q S_n = a_1 q + a_1 q^2 + a_1 q^3 + \dots + a_1 q^{n-1} + a_1 q^n \quad (2)$$

Subtraindo (I) de (II), temos:

$$S_n - q S_n = a_1 - a_1 q^n$$

Colocando em evidência, chega-se em:

$$S_n = a_1 \frac{(q^n - 1)}{(q - 1)}$$

Sendo assim, nosso somatório é igual a:

$$S_{\log_2 n + 1} = 1 \frac{\left(\left(\frac{3}{2}\right)^{\log_2 n + 1} - 1\right)}{\left(\left(\frac{3}{2}\right) - 1\right)}$$

Somando 1 no numerador e 1 no denominador, temos:

$$S_{\log_2 n + 1} = \frac{\left(\frac{3}{2}\right)^{\log_2 n + 1}}{\left(\frac{3}{2}\right)} = \left(\frac{3}{2}\right)^{\log_2 n} = n^{\log_2 3/2}$$

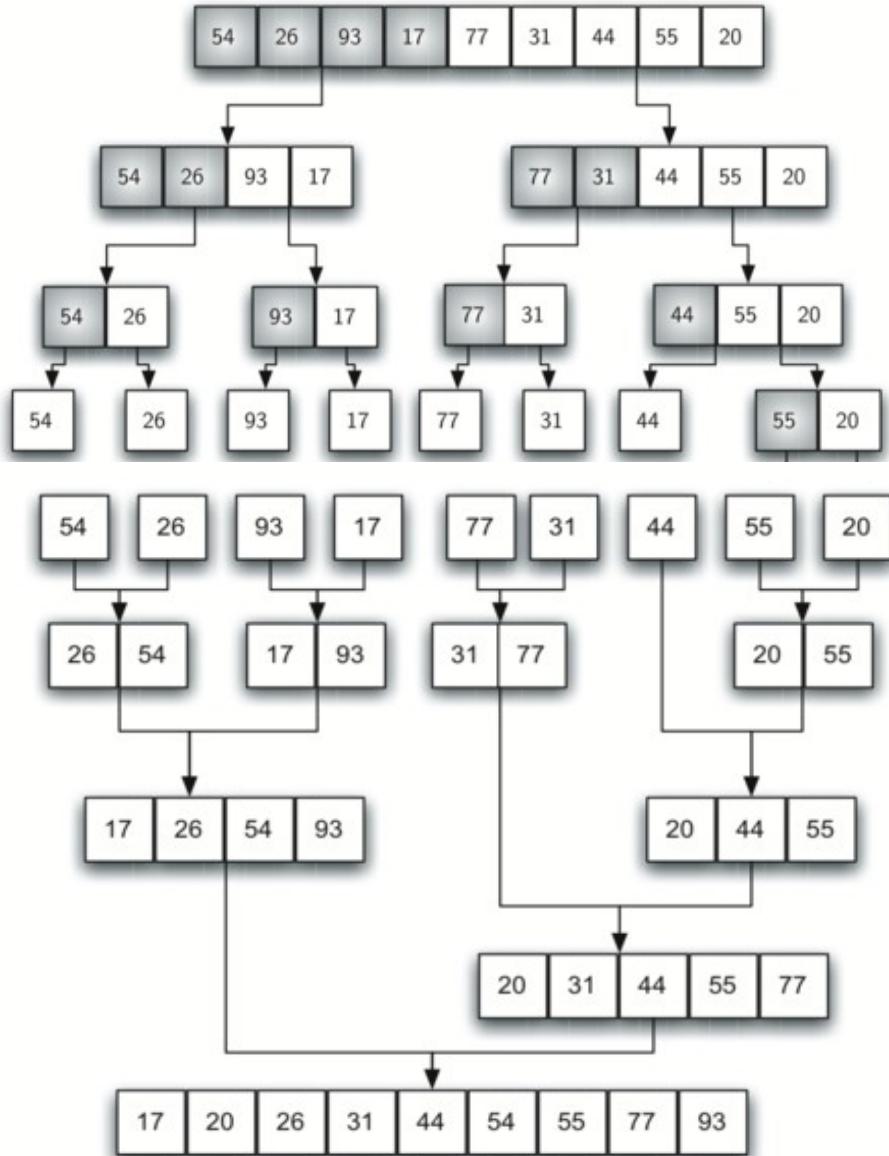
Assim, a função $T(n)$ fica:

$$T(n) = c n n^{\log_2 3/2} = c n n^{\log_2 3 - \log_2 2} = c n n^{\log_2 3 - 1} = c n n^{\log_2 3} n^{-1} = c n^{\log_2 3}$$

Portanto, a complexidade do algoritmo de Karatsuba é $O(n^{\log_2 3})$, que é igual a $O(n^{1.585})$. Note com a estratégia dividir para conquistar conseguimos reduzir a complexidade da multiplicação de inteiros.

O algoritmo Mergesort

Trata-se de um algoritmo recursivo que emprega a estratégia dividir para conquistar em problemas de ordenação. Se a lista estiver vazia ou tiver um único elemento, ela está ordenada por definição (o caso base). Se a lista tiver mais de um elemento, dividimos a lista e invocamos recursivamente um Mergesort em ambas as metades. Assim que as metades estiverem ordenadas, a operação fundamental, chamada de **intercalação**, é realizada. Intercalar é o processo de pegar duas listas menores ordenadas e combiná-las de modo a formar uma lista nova, única e ordenada. A figura a seguir ilustra as duas fases principais do algoritmo Mergesort: a divisão e a intercalação.



A seguir apresentamos uma função em Python para implementar o algoritmo MergeSort.

```

# Implementação em Python do algoritmo de ordenação MergeSort
def MergeSort(L):

    if len(L) > 1:

        meio = len(L)//2
        LE = L[:meio]    # Lista Esquerda
        LD = L[meio:]   # Lista Direita

        # Aplica recursivamente nas sublistas
        MergeSort(LE)
        MergeSort(LD)

        # Quando volta da recursão inicia aqui!
        i, j, k = 0, 0, 0
        # Faz a intercalação das duas listas (merge)
        while i < len(LE) and j < len(LD):
            if LE[i] < LD[j]:
                L[k] = LE[i]
                i += 1
            else:
                L[k] = LD[j]
                j += 1
            k += 1

        while i < len(LE):
            L[k] = LE[i]
            i += 1
            k += 1

        while j < len(LD):
            L[k] = LD[j]
            j += 1
            k += 1

```

Os três passos úteis dos algoritmos de dividir para conquistar que se aplicam ao MergeSort são:

1. Dividir: Calcula o ponto médio do sub-arranjo, o que demora um tempo constante $O(1)$;
2. Conquistar: Recursivamente resolve dois subproblemas, cada um de tamanho $n/2$, o que contribui com $T(n/2) + T(n/2)$ para o tempo de execução;
3. Combinar: Unir os sub-arranjos em um único conjunto ordenado, que leva o tempo $O(n)$;

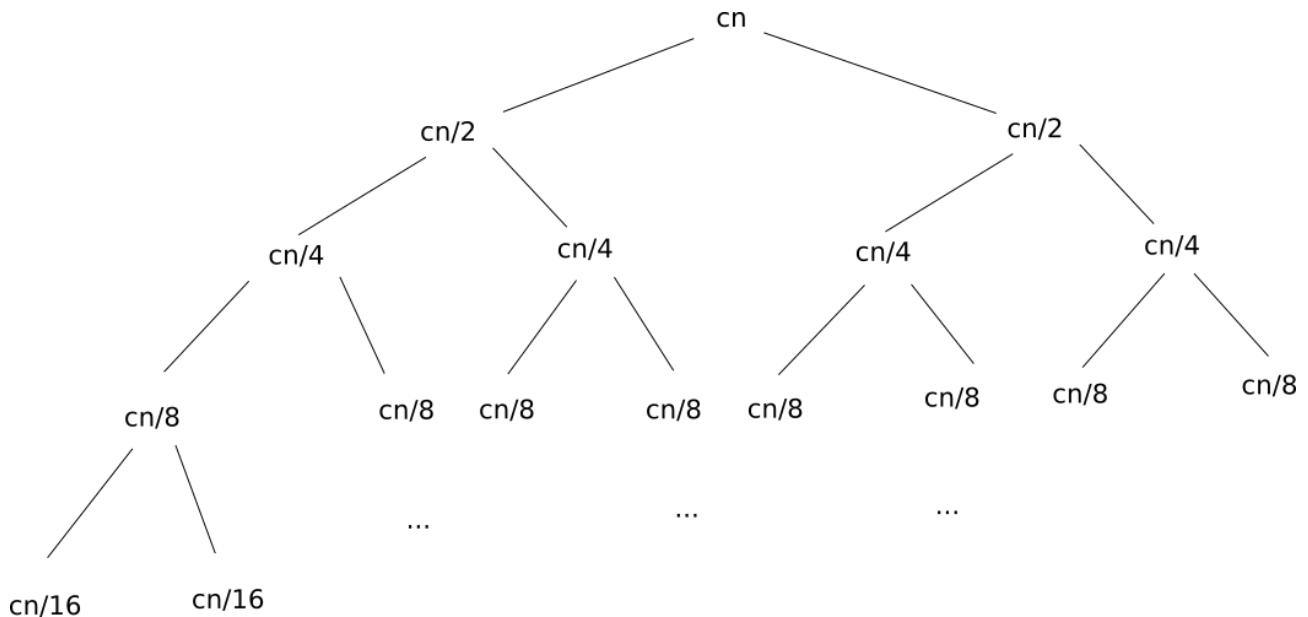
Assim, podemos escrever a relação de recorrência como:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + O(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Qual é a complexidade desse algoritmo?

A seguir empregamos o método da árvore de recursão para descobrir a resposta.

A árvore de recursão para a recorrência acima é dada por:



Podemos ver que em nós folhas devemos ter custo constante, ou seja, $\frac{n}{2^k}=1$, o que implica em $k=\log_2 n$ como a altura da árvore de recursão. Note que ao somar os custos em cada nível desta árvore temos um valor constante igual a cn . Sendo assim, A função T(n) fica:

$$T(n)=cn+cn+cn+\dots+cn=\sum_{i=0}^{\log_2 n} cn=cn\sum_{i=0}^{\log_2 n} 1=cn\log_2 n$$

o que mostra que o algoritmo Mergesort possui complexidade $O(n \log_2 n)$.

O Problema da Multiplicação de Matrizes

Sabemos que multiplicar matrizes é uma operação recorrente em diversas aplicações da computação, matemática, estatística e áreas afins. Nesta seção, apresentaremos um algoritmo mais eficiente que a abordagem tradicional utilizando a estratégia dividir para conquistar. Primeiramente, lembre que o produto matricial é definido como:

$$\begin{array}{ccc}
 & \vec{b}_1 & \vec{b}_2 \\
 & \downarrow & \downarrow \\
 \vec{a}_1 \rightarrow & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot & \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix} \\
 \vec{a}_2 \rightarrow & &
 \end{array}$$

A B C

ou seja, os elemento da matriz C são os produtos escalares entre as linhas de A e as colunas de B. Em outras palavras, temos:

$$c_{ij} = \vec{a}_i \cdot \vec{b}_j = \sum_{k=1}^n a_{ik} b_{kj}$$

O algoritmo tradicional para multiplicar duas matrizes quadradas A e B é ilustrado a seguir.

MATRIX-MULTIPLY(A, B, C, n)

```

1  for  $i = 1$  to  $n$                                 // compute entries in each of  $n$  rows
2    for  $j = 1$  to  $n$                       // compute  $n$  entries in row  $i$ 
3      for  $k = 1$  to  $n$ 
4         $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$  // add in another term of equation (4.1)

```

Pode-se ver claramente, que como temos 3 estruturas de repetição FOR aninhadas, a complexidade dessa função é $O(n^3)$.

Uma simples estratégia dividir para conquistar consiste em assumir sem perda de generalidade que $n=2^m$ (potência de 2) e então subdividir as matrizes A e B em 4 submatrizes de tamanho $n/2 \times n/2$. Dessa forma, as matrizes de entrada A e B podem ser decompostas como:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

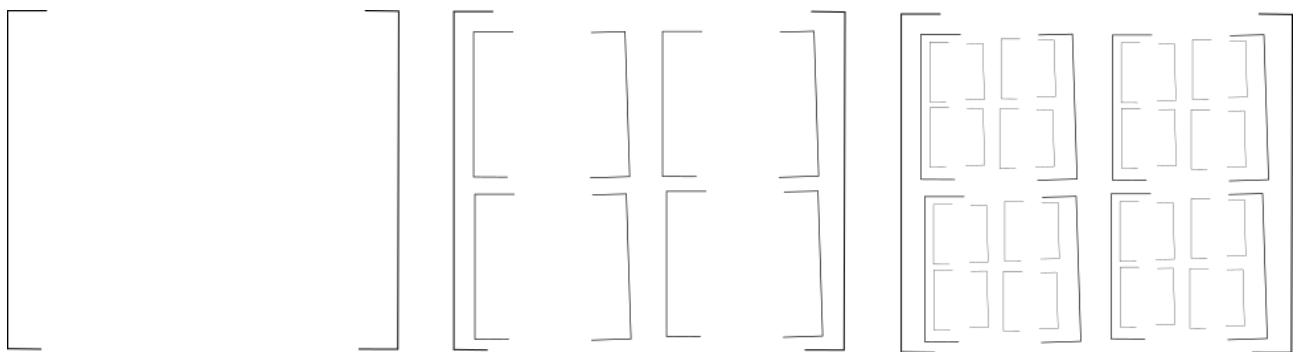
Dessa forma, o produto matricial fica dado por:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Note que essa operação envolve:

8 multiplicações de matrizes $n/2 \times n/2$ + 4 adições de matrizes $n/2 \times n/2$

A ideia consiste em recursivamente subdividir a matriz até atingir o caso base em que temos matrizes de tamanho 1×1 .



1 matriz 4×4

4 matrizes 2×2

16 matrizes 1×1

O algoritmo a seguir implementa essa estratégia dividir para conquistar.

MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

```

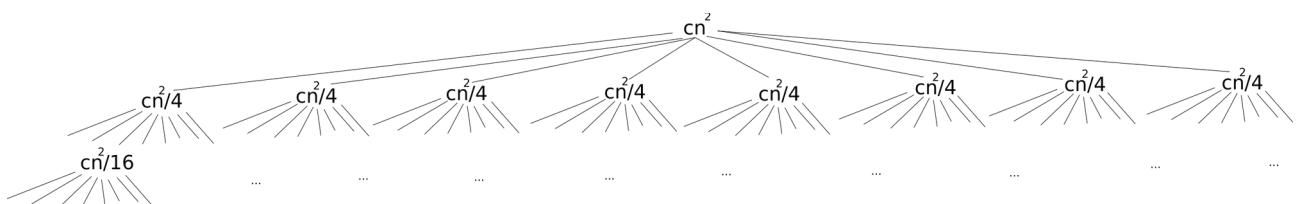
1  if  $n == 1$ 
2  // Base case.
3       $c_{11} = c_{11} + a_{11} \cdot b_{11}$ 
4      return
5  // Divide.
6  partition  $A$ ,  $B$ , and  $C$  into  $n/2 \times n/2$  submatrices
     $A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22};$ 
    and  $C_{11}, C_{12}, C_{21}, C_{22}$ ; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE( $A_{11}, B_{11}, C_{11}, n/2$ )
9  MATRIX-MULTIPLY-RECURSIVE( $A_{11}, B_{12}, C_{12}, n/2$ )
10 MATRIX-MULTIPLY-RECURSIVE( $A_{21}, B_{11}, C_{21}, n/2$ )
11 MATRIX-MULTIPLY-RECURSIVE( $A_{21}, B_{12}, C_{22}, n/2$ )
12 MATRIX-MULTIPLY-RECURSIVE( $A_{12}, B_{21}, C_{11}, n/2$ )
13 MATRIX-MULTIPLY-RECURSIVE( $A_{12}, B_{22}, C_{12}, n/2$ )
14 MATRIX-MULTIPLY-RECURSIVE( $A_{22}, B_{21}, C_{21}, n/2$ )
15 MATRIX-MULTIPLY-RECURSIVE( $A_{22}, B_{22}, C_{22}, n/2$ )

```

Note que o particionamento das matrizes A , B e C podem ser realizados com complexidade $O(n^2)$, o que nos permite escrever a seguinte recorrência:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

Utilizando o método da árvore de recursão temos a seguinte construção.



Podemos ver que em nós folhas devemos ter custo constante, ou seja, $\frac{n}{2^k}=1$, o que implica em $k=\log_2 n$ como a altura da árvore de recursão. O custo de cada nível da árvore é dado por:

Nível 0: cn^2

Nível 1: $8 cn^2/4 = 2 cn^2$

Nível 2: $64 cn^2/16 = 4 cn^2$

...

Dessa forma, temos:

$$T(n) = cn^2 + 2cn^2 + 4cn^2 + 8cn^2 + \dots = \sum_{i=0}^{\log_2 n} 2^i cn^2 = cn^2 \sum_{i=0}^{\log_2 n} 2^i$$

Lembre-se que $2^{i+1} = 2 \cdot 2^i = 2^i + 2^i$, o que nos leva a $2^i = 2^{i+1} - 2^i$. Dessa forma, temos:

$$T(n) = cn^2 \sum_{i=1}^{\log_2 n} 2^{i+1} - 2^i = cn^2 (2^{\log_2 n+1} - 2^0) = cn^2 (2 \cdot 2^{\log_2 n} - 1) = cn^2 (2n - 1) = 2cn^3 - cn^2$$

Em outras palavras, o método com a estratégia dividir para conquistar tem a mesma complexidade do que o método padrão. A seguir veremos como podemos melhorar essa estratégia através do algoritmo de Strassen.

O algoritmo de Strassen

A partir do particionamento das matrizes de entrada A e B, devemos aqui realizar o calculo de 10 matrizes (S_1, S_2, \dots, S_{10}). Porém, para o cálculo dessas matrizes, utilizamos apenas soma e subtração de matrizes, conforme segue abaixo:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

As 10 matrizes são definidas como:

$$\begin{array}{ll} S_1 = B_{12} - B_{22} & S_6 = B_{11} + B_{22} \\ S_2 = A_{11} - A_{12} & S_7 = A_{12} - A_{22} \\ S_3 = A_{21} + A_{22} & S_8 = B_{21} + B_{22} \\ S_4 = B_{21} - B_{11} & S_9 = A_{11} - A_{21} \\ S_5 = A_{11} + A_{22} & S_{10} = B_{11} + B_{12} \end{array}$$

Note que o custo computacional de calcular todas essas 10 matrizes é $O(n^2)$.

Em seguida, devemos calcular 7 matrizes (envolve produto matricial):

$$\begin{aligned} P_1 &= A_{11}S_1 = A_{11}B_{12} - A_{11}B_{22} \\ P_2 &= S_2B_{22} = A_{11}B_{22} - A_{12}B_{22} \\ P_3 &= S_3B_{11} = A_{21}B_{11} + A_{22}B_{11} \\ P_4 &= A_{22}S_4 = A_{22}B_{21} - A_{22}B_{11} \\ P_5 &= S_5S_6 = A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\ P_6 &= S_7S_8 = A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \\ P_7 &= S_9S_{10} = A_{11}B_{11} + A_{11}B_{12} - A_{21}B_{11} - A_{21}B_{12} \end{aligned}$$

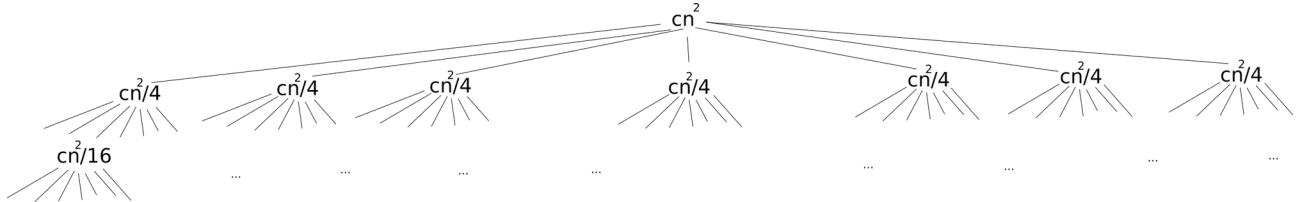
Pode-se mostrar que as 4 submatrizes C_{11}, C_{12}, C_{21} e C_{22} podem ser expressas como:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= P_1 + P_2 = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= P_3 + P_4 = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= P_5 + P_1 - P_3 - P_7 = A_{22}B_{22} + A_{21}B_{12} \end{aligned}$$

Assim, a recorrência para o algoritmo de Strassen é dada por:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

o que gera a seguinte árvore de recursão:



Podemos ver que em nós folhas devemos ter custo constante, ou seja, $\frac{n}{2^k} = 1$, o que implica em $k = \log_2 n$ como a altura da árvore de recursão. O custo de cada nível da árvore é dado por:

Nível 0: cn^2

Nível 1: $7 cn^2/4$

Nível 2: $49 cn^2/16 = (7/4)^2 cn^2$

Nível 3: $(7/4)^3 cn^2$

...

Sendo assim, podemos escrever $T(n)$ como:

$$T(n) = cn^2 + \left(\frac{7}{4}\right)cn^2 + \left(\frac{7}{4}\right)^2 cn^2 + \left(\frac{7}{4}\right)^3 cn^2 + \dots + \left(\frac{7}{4}\right)^{\log_2 n} cn^2$$

o que nos leva a:

$$T(n) = cn^2 \sum_{i=0}^{\log_2 n} \left(\frac{7}{4}\right)^i$$

Temos uma P.G. com $1 + \log_2 n$ termos em que o primeiro termo $a_1 = 1$ e $q = \frac{7}{4}$. Sendo assim, o somatório vale:

$$S = 1 \cdot \frac{\left(\left(\frac{7}{4}\right)^{1+\log_2 n} - 1\right)}{\left(\frac{7}{4} - 1\right)}$$

Somando 1 no numerador e no denominador, chega-se a:

$$S = \frac{\left(\frac{7}{4}\right)^{1+\log_2 n}}{\frac{7}{4}} = \left(\frac{7}{4}\right)^{\log_2 n}$$

Pela propriedade $a^{\log_c b} = c^{\log_b a}$, podemos escrever:

$$S = n^{\log_2(7/4)} = n^{\log_2 7 - 2} = n^{\log_2 7} n^{-2}$$

Portanto, voltando a $T(n)$, chegamos em:

$$T(n) = cn^2 n^{\log_2 7} n^{-2} = cn^{\log_2 7}$$

o que mostra que o algoritmo de Strassen tem complexidade $O(n^{\log_2 7})$, o que equivale a aproximadamente $O(n^{2.8})$. Portanto, o algoritmo de Strassen é capaz de melhorar o desempenho assintótico da multiplicação de matrizes.

O problema do par de pontos mais próximos

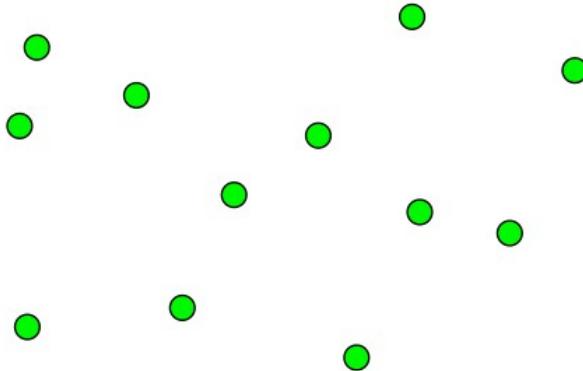
Suponha que sejam dados n pontos no plano, o objetivo consiste em encontrar o par composto pelos pontos mais próximos entre si. Esse problema é muito importante pois define uma primitiva muito comum em diversos problemas em geometria computacional e computação gráfica. Iniciaremos apresentando as variáveis e a notação envolvidas no problema.

Seja $P = \{p_1, p_2, \dots, p_n\}$ um conjunto de n pontos no plano em que cada p_i é definido em termos de suas coordenadas (x_i, y_i) . Podemos associar para cada par de pontos $p_i, p_j \in P$, a distância Euclidiana entre eles, denotada aqui por $d(p_i, p_j)$ e definida como:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

O objetivo do problema consiste em encontrar os pontos p_i, p_j que minimizam a distância entre eles, ou seja:

$$p_1^*, p_2^* = \arg \min \{d(p_i, p_j)\}$$



Por motivos didáticos, vamos iniciar com o caso 1D. Neste caso, os pontos estão localizados na reta real e possuem apenas uma coordenada x_i . A solução trivial para o problema envolve ordenar os pontos pelas coordenadas x_i e percorrer a lista calculando a distância de cada ponto para o seu sucessor na lista.

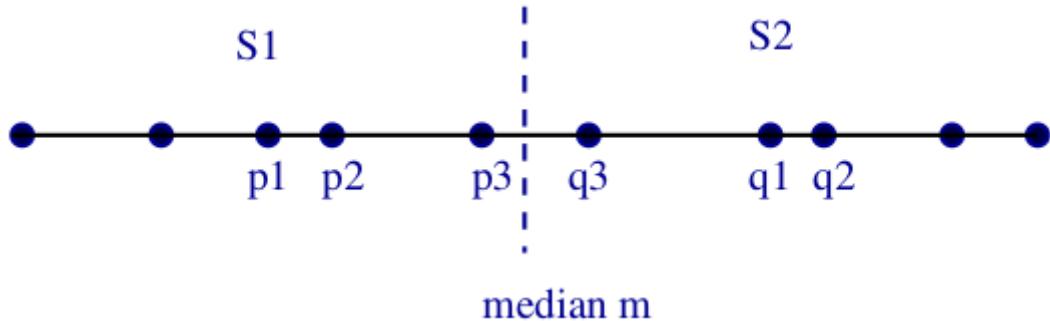
```
ClosestPair_1D(L) {
    // L é a lista das coordenadas dos pontos
    n = len(L)
    P = argsort(L) // P armazena índices dos pontos após ordenação
    sort(L)
```

```

min_d = ∞
pto1 = -1
pto2 = -1
for i = 1 to n-1 {
    d = abs(L[i] - L[i+1])
    if d < min_d {
        min_d = d
        pto1 = P[i]
        pto2 = P[i+1]
    }
}
return (pto1, pto2)
}

```

Note que a complexidade desse algoritmo é $O(n \log n)$ que é o limite mínimo para a ordenação. É possível melhorar esse algoritmo? A resposta é não. Mesmo utilizando a estratégia dividir para conquistar para quebrar o problema em 2 subproblemas, a complexidade continua sendo $O(n \log n)$.



No caso 2D, a solução trivial é a força bruta, ou seja, consiste em calcular as distâncias entre todos os possíveis pares de pontos e verificar qual delas é a menor.

```

ClosestPair_2D(L) {
    // L é a lista das coordenadas dos pontos
    n = len(L)
    min_d = ∞
    pto1 = -1
    pto2 = -1
    for i = 1 to n {
        for j = i to n {
            Calculate the distance d between pi and pj
            if d < min_d {
                min_d = d
                pto1 = i
                pto2 = j
            }
        }
    }
    return (pto1, pto2)
}

```

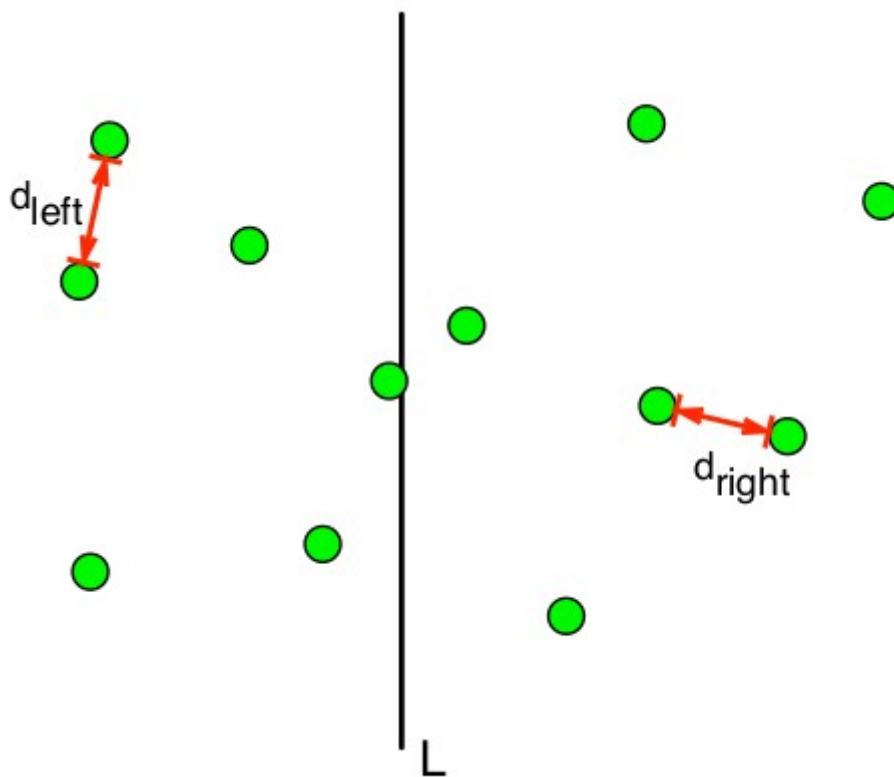
Note que devermos calcular o seguinte número de distâncias:

$$T(n) = n + (n-1) + (n-2) + \dots + 1 = \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Portanto, o algoritmo tem complexidade $O(n^2)$. A pergunta natural é: é possível melhorar esse algoritmo? A resposta é sim! No caso 2D podemos desenvolver um algoritmo log-linear. A seguir veremos como isso é possível. A ideia consiste em adotar a estratégia dividir para conquistar da seguinte maneira:

1. Encontrar o par mais próximo na metade a esquerda do plano
2. Encontrar o par mais próximo na metade a direita do plano
3. Combinar as soluções verificando se existe algum par formado por um ponto de Q e outro de R

A divisão, ou seja, a decomposição do problema em dois subproblemas menores pode ser realizada criando uma lista P_x dos pontos ordenados por ordem crescente de coordenada x. A mediana M desse conjunto define uma reta que partitiona o plano em duas regiões: Q (left) e R (right). Recursivamente, podemos encontrar os pares mais próximos em cada região. A figura a seguir ilustra esse processo.

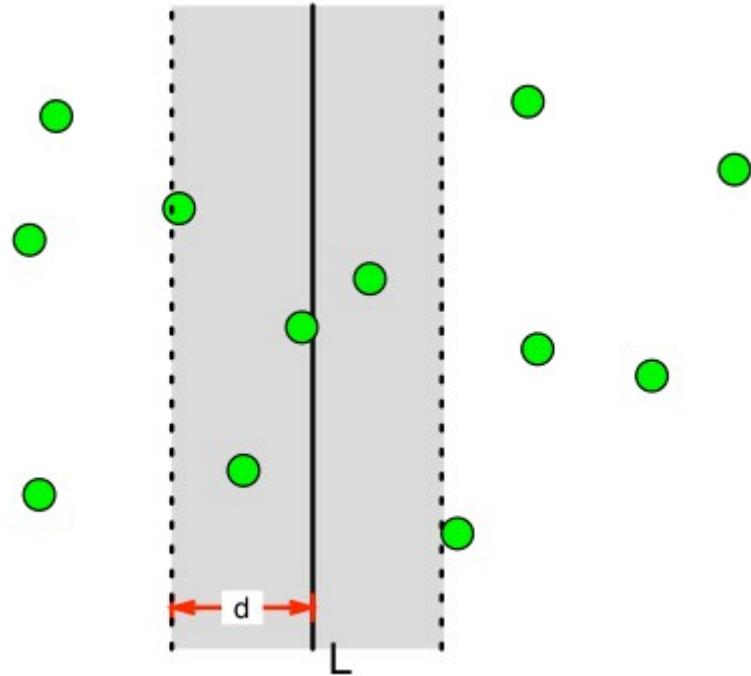


O passo da conquista consiste em combinar as soluções dos subproblemas menores. Seja:

$$d = \min\{d_{left}, d_{right}\}$$

ou seja d é a menor das menores distâncias em Q e R. A princípio, d seria a solução do problema maior, mas note que pode existir um par em que um dos pontos está em Q e outro está em R, cuja distância entre eles é menor que d . Em outras palavras, o resultado a seguir mostra algo muito interessante.

Teorema: Se existe um par $\{q, r\}$ com $q \in Q$ e $r \in R$ (ou vice-versa) tal que $d(q, r) < d$, então ambos q e r devem estar no máximo a uma distância d de L (numa faixa ao redor de L).



Prova:

Podemos provar a afirmação acima notando que a reta L é definida pelo ponto x^* com coordenada x igual a mediana M . Sabemos que as coordenadas dos pontos são: $q = (q_x, q_y)$ e $r = (r_x, r_y)$. Pela definição de x^* , temos:

$$q_x \leq x^* \leq r_x \quad (\text{I})$$

Isso implica que $x^* \leq r_x$. Subtraindo q_x de ambos os lados, temos:

$$x^* - q_x \leq r_x - q_x \leq d(q, r) < d$$

De modo similar, de (I) temos que $x^* \geq q_x$. Isso implica que $-x^* \leq -q_x$. Somando r_x de ambos os lados, temos:

$$r_x - x^* \leq r_x - q_x \leq d(q, r) < d$$

Portanto, os pontos de interesse estão dentro da margem definida por d .

A seguir veremos um outro resultado fundamental para o problema em questão.

Teorema: Seja S_y uma lista com as coordenadas y de todos os pontos a uma distância d de L (na faixa) em ordem crescente. Suponha que $S_y = \{p_1, p_2, \dots, p_m\}$ com $m \leq n$. Se $d(p_i, p_j) < d$, então $j - i \leq 15$. Em outras palavras, se dois pontos estão próximos na faixa, então eles também estão próximos na lista S_y .

Prova:

1. Divida a região dentro da faixa em quadrados de lado $d/2$



Então, quantos pontos podem existir em cada quadrado? Apenas 1, pois senão existiriam 2 pontos cuja distância entre eles seria menor que d .

2. Suponha agora que 2 pontos são separados por 15 índices. Então, existem 3 linhas separando os pontos. Como cada linha tem lado $d/2$, a distância entre os pontos é maior ou igual a $3d/2$. Ou seja, a distância entre os pontos é maior que d .

3. Portanto, basta procurar no máximo 15 posições a frente em S_y .

Obs: Note que o valor 15 pode ser diminuído, mas o que importa aqui é que ele é uma constante!

O algoritmo a seguir mostra a abordagem dividir para conquistar para o problema do par mais próximo no \mathbb{R}^2 .

```

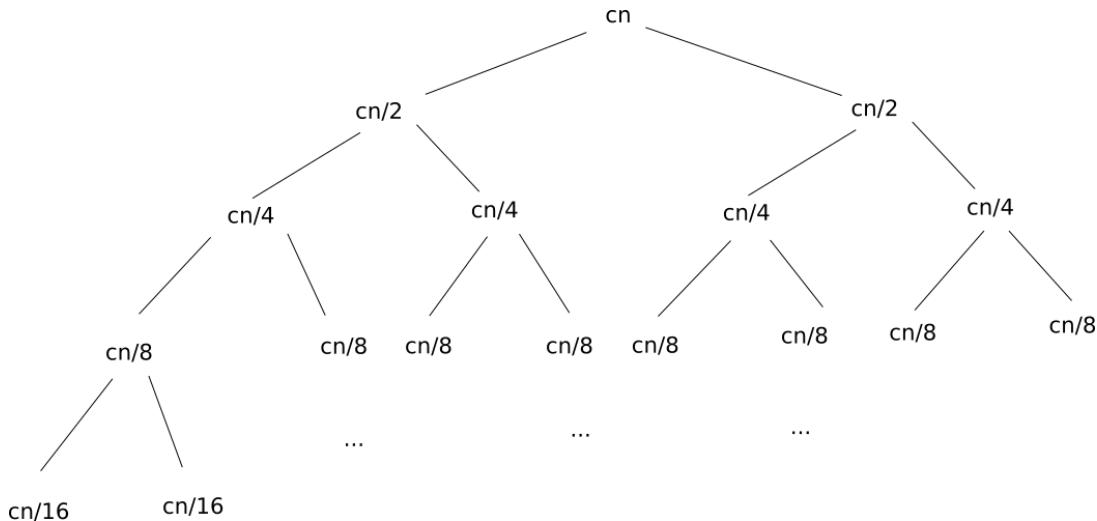
ClosestPair_DC_2D(Px, Py) {
    // Px são as coordenadas x dos pontos
    // Py são as coordenadas y dos pontos
    // caso base (apenas 2 pontos)
    if |Px| == 2
        return dist(Px[1], Px[2], Py[1], Py[2])
    // Dividir
    d1 = ClosestPair_DC_2D(FirstHalf(Px, Py))
    d2 = ClosestPair_DC_2D(SecondHalf(Px, Py))
    d = min(d1, d2)
    // Conquistar (merge)
    Sx = points in Px in the narrow band around L
    Sy = points in Py in the narrow band around L
    for i = 1 to |Sy| {
        m = min(i+15, |Sy|)
        for j = i to m
            d = min(dist(Sx[i], Sx[j], Sy[i], Sy[j]), d)
    }
    return d
}

```

Note que trata-se de um algoritmo recursivo que quebra o problema de tamanho n em 2 subproblemas menores de tamanho $n/2$ e depois combina as soluções (merge) com custo linear em n , ou seja, com complexidade $O(n)$, pois o loop mais interno é constante. Sendo assim, a recorrência é dada por:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos expandir a árvore de recursão.



Sabemos que o nível máximo da árvore ocorre quando $\frac{n}{2^k}=1$, ou seja o número de níveis da árvore é $k=\log_2 n$. Somando os custos em cada nível da árvore, temos o custo total do algoritmo, que é dado por:

$$T(n) = \sum_{i=1}^{\log_2 n} cn = cn \log_2 n$$

o que é igual a $O(n \log_2 n)$ e portanto melhor que o método trivial que é $O(n^2)$.

A Transformada de Fourier Discreta

O processamento digital de sinais lida com sinais no domínio do tempo. Sinais analógicos e digitais estão presentes no mundo ao nosso redor: desde a variação do preço do dólar nos últimos anos até a sinais de voz humana em gravações telefônicas, todos esses tipos de dados são sinais que variam no tempo. Durante a análise de tais sinais, é fundamental ter acesso as componentes de frequência dos mesmos, visto que ao modificar determinadas frequências dos sinais podemos por exemplo filtrar ruídos ou até mesmo reforçar agudos ou graves.

- A Transformada de Fourier é a ferramenta matemática que nos permite analisar as componentes de frequência de sinais.

- No caso de sinais discretos (processamento digital), utiliza-se a versão discreta, conhecida como DFT (Discrete Fourier Transform).

Ideia: mapear n pontos do domínio do tempo para n pontos no domínio da frequência.

Seja $x[]$ um sinal de entrada com n amostras. Então a DFT de $x[]$, denotada por $X[]$, é dada por:

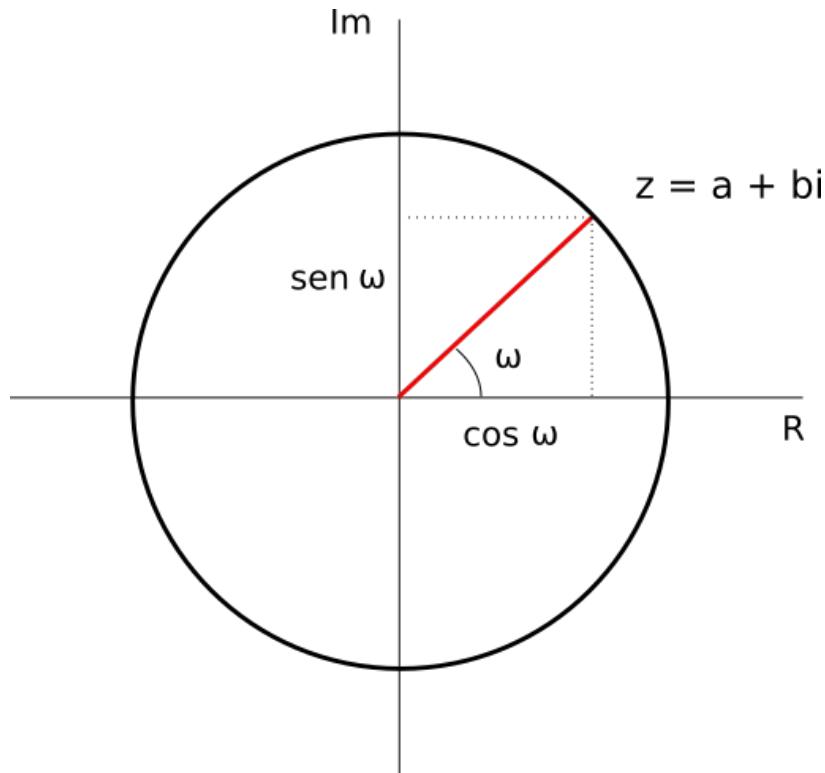
$$X[k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{-\left(\frac{j2\pi kn}{N}\right)} \quad \text{para } k = 0, 1, 2, \dots, N - 1$$

Analogamente, a transformada inversa, conhecida por IDFT (Inverse Discrete Fourier Transform), que reconstrói o sinal original a partir da sua representação no domínio da frequência, é dada por:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\frac{j2\pi kn}{N}} \quad \text{para } n = 0, 1, 2, \dots, N - 1$$

A seguir, iremos ver que tanto a DFT quanto a IDFT são nada mais nada menos que produtos entre uma matriz quadrada de coeficientes e um vetor com as amostras do sinal.

Uma pergunta relevante é: porque a DFT decompõe um sinal em componentes de frequência? Lembre-se da relação entre exponenciais complexas e as funções trigonométricas seno e cosseno. A figura a seguir ilustra a circunferência no plano complexo de Argand-Gauss.



Note que um número complexo, além de sua forma padrão, pode ser expresso pela notação polar, em termos de sua magnitude e do ângulo ω (fase):

$$z = \rho e^{iw}$$

onde $\rho = \sqrt{a^2 + b^2}$. Como estamos na circunferência trigonométrica, $a = \cos \omega$ e $b = \sin \omega$. Além disso, note que, neste caso, devido a identidade trigonométrica, temos que:

$$\rho = \sqrt{\sin^2(\omega) + \cos^2(\omega)} = 1$$

de modo que

$$e^{iw} = \cos(\omega) + i \sin(\omega)$$

Se conjugarmos o número complexo, ou seja, trocarmos o sinal da parte imaginária, temos:

$$e^{-iw} = \cos(\omega) - i \sin(\omega)$$

Na engenharia e na computação, por questões de nomenclatura, é comum usar a letra j ao invés de i para indicar a unidade complexa, ou seja, $j = \sqrt{-1}$. Portanto, note que:

$$e^{\frac{-j2\pi kn}{N}} = \cos\left(\frac{2\pi kn}{N}\right) - j \sin\left(\frac{2\pi kn}{N}\right)$$

Assim, o que a DFT faz é essencialmente escrever um sinal $x[]$ do domínio do tempo como combinação linear de senos e cossenos (frequências do sinal). Porém, note que o resultado da DFT de um sinal é um vetor de mesmo tamanho composto por números complexos (e não reais)! Por essa razão, é comum analisarmos a magnitude dos coeficientes da DFT, uma vez que cada valor é composto de uma parte real e outra imaginária.

A Matriz DFT

A DFT é aplicada fazendo $\vec{X} = W \vec{x}$, onde W é a matriz dos coeficientes da DFT:

$$W_{k,n} = e^{\frac{-j2\pi kn}{N}} \quad \text{para } k = 0, 1, 2, \dots, N-1 \text{ e } n = 0, 1, 2, \dots, N-1$$

Note que se $k = 0$ ou $n = 0$, $W_{k,n} = e^0 = 1$. Na prática, isso significa que todos os elementos da primeira linha e da primeira coluna de W são iguais a 1. Em todas as demais posições da matriz, os elementos são potências de w , definido como:

$$w = e^{\frac{-j2\pi}{N}}$$

Por exemplo, o elemento da posição $(2, 2)$ é dado por:

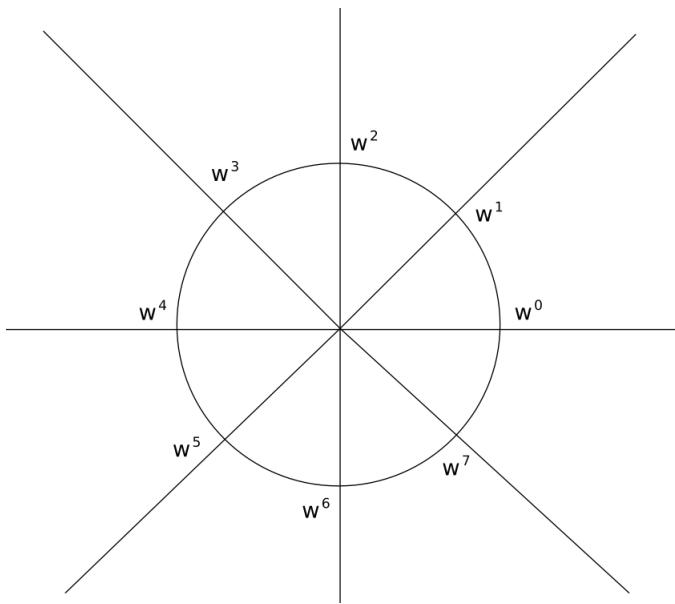
$$W_{2,2} = e^{\frac{-j2\pi \cdot 2 \cdot 2}{N}} = e^{\frac{-j2\pi \cdot 4}{N}} = e^{\frac{-j2\pi}{N}} e^{\frac{-j2\pi}{N}} e^{\frac{-j2\pi}{N}} e^{\frac{-j2\pi}{N}} = w w w w = w^4$$

Dessa forma, pode-se mostrar que a matriz DFT W é uma matriz de Vandermonde, dada por:

$$W = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & w^3 & \dots & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \dots & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & \dots & w^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & \dots & w^{(N-1)(N-1)} \end{bmatrix}$$

Ex: Calcule a DFT do sinal $x[] = [1, 2, 3, 4, 1, 2, 3, 4]$.

Primeiramente, como $N = 8$, devemos dividir a circunferência unitária no plano complexo em 8 intervalos iguais, conforme ilustra a figura a seguir.



$$\text{onde } w = e^{-j2\frac{\pi}{8}} = \cos \frac{\pi}{4} - j \sin \frac{\pi}{4} = \frac{\sqrt{2}}{2} - j \frac{\sqrt{2}}{2}$$

A matriz W para esse caso fica:

$$W = \frac{1}{\sqrt{8}} \begin{bmatrix} w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ w^0 & w^2 & w^4 & w^6 & w^8 & w^{10} & w^{12} & w^{14} \\ w^0 & w^3 & w^6 & w^9 & w^{12} & w^{15} & w^{18} & w^{21} \\ w^0 & w^4 & w^8 & w^{12} & w^{16} & w^{20} & w^{24} & w^{28} \\ w^0 & w^5 & w^{10} & w^{15} & w^{20} & w^{25} & w^{30} & w^{35} \\ w^0 & w^6 & w^{12} & w^{18} & w^{24} & w^{30} & w^{36} & w^{42} \\ w^0 & w^7 & w^{14} & w^{21} & w^{28} & w^{35} & w^{42} & w^{49} \end{bmatrix}$$

O preenchimento da matriz W é feito linha a linha, efetuando deslocamentos variáveis na circunferência unitária. Para a primeira linha, o deslocamento é zero e temos todos os elementos iguais a w^0 . Na segunda linha o deslocamento é de um intervalo, na terceira linha o deslocamento é de 3 intervalos e assim sucessivamente. Em outras palavras, isso significa que $w^{N+n} = w^n$. Sendo assim, a matriz simplifica-se para:

$$W = \frac{1}{\sqrt{8}} \begin{bmatrix} w^0 & w^0 \\ w^0 & w^1 & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ w^0 & w^2 & w^4 & w^6 & w^0 & w^2 & w^4 & w^6 \\ w^0 & w^3 & w^6 & w^1 & w^4 & w^7 & w^2 & w^5 \\ w^0 & w^4 & w^0 & w^4 & w^0 & w^4 & w^0 & w^4 \\ w^0 & w^5 & w^2 & w^7 & w^4 & w^1 & w^6 & w^3 \\ w^0 & w^6 & w^4 & w^2 & w^0 & w^6 & w^4 & w^2 \\ w^0 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w^1 \end{bmatrix} = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & -j & -jw & -1 & -w & j & jw \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -jw & j & w & -1 & jw & -j & -w \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -w & -j & jw & -1 & w & j & -jw \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & jw & j & -w & -1 & -jw & -j & w \end{bmatrix}$$

uma vez que calculando os valores de w^0 até w^7 temos:

$$\begin{aligned}
 w^0 &= 1 \\
 w^1 &= w \\
 w^2 &= \left(\frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}\right)\left(\frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}\right) = \frac{1}{2} - \frac{1}{2}j - \frac{1}{2}j + \frac{1}{2}j^2 = -j \\
 w^3 &= -jw \\
 w^4 &= (-j)(-j) = j^2 = -1 \\
 w^5 &= -w \\
 w^6 &= j \\
 w^7 &= jw
 \end{aligned}$$

A função a seguir implementa a DFT de um sinal x de N amostras. Note que temos um produto entre a matriz W definida pelo algoritmo e o sinal de entrada x .

```

DFT(x, N) {
    Define the DFT matrix W
    for i = 0 to N - 1 {
        X[i] = 0
        for j = 0 to N - 1
            X[i] = X[i] + W[i, j]*x[j]
    }
    return X
}

```

A construção da matriz W tem complexidade $O(N^2)$ e os dois loops FOR aninhados também, logo é fácil perceber que a complexidade da DFT é quadrática. Para sinais com um grande número de amostras, o processo torna-se bastante lento. Para acelerar o método, foi proposto o algoritmo FFT (Fast Fourier Transform), utilizando uma estratégia dividir para conquistar.

Note que o resultado da DFT é um vetor de números complexos. Para visualizar a DFT de um sinal, podemos plotar a magnitude dos elementos de X , ou seja:

$$\|X[i]\| = \sqrt{\Re\{X[i]\}^2 + \Im\{X[i]\}^2} \quad \text{para } i = 1, 2, \dots, N$$

Também é possível visualizar a fase da DFT, que nada mais é que o ângulo que o número complexo forma com o eixo R no plano complexo:

$$\theta[i] = \arctan\left(\frac{\Re\{X[i]\}}{\Im\{X[i]\}}\right) \quad \text{para } i = 1, 2, \dots, N$$

O algoritmo Fast Fourier Transform (FFT)

- Algoritmo eficiente para o cálculo da DFT
- Estratégia dividir para conquistar

A FFT explora as simetrias de $e^{\frac{-j2\pi kn}{N}}$. Seja $W_N = e^{\frac{-j2\pi}{N}}$, então temos $W_N^k = e^{\frac{-j2\pi k}{N}}$.

- i) Simetria do conjugado complexo

$$W_N^{k(N-n)} = W_N^{kN} W_N^{-kn} = W_N^{-kn} = (W_N^{kn})^*$$

ii) Periodicidade (devido a circunferência trigonométrica)

$$W_N^{kn} = 1 \quad W_N^{kn} = W_N^{kN} W_N^{kn} = W_N^{k(N+n)}$$

FFT por decimação no tempo

Objetivo: construir uma grande DFT a partir de várias DFT menores.

Sem perda de generalidade, iremos assumir que $N=2^m$, para $m > 1$ (se o número de amostras do sinal não é potência de 2, podemos completar com zeros até a próxima potência de 2).

Ideia: Separar as amostras de $x[n]$ em índices pares e ímpares.

$$X[k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n] W_N^{kn} = \frac{1}{\sqrt{N}} \sum_{n \text{ par}} x[n] W_N^{kn} + \frac{1}{\sqrt{N}} \sum_{n \text{ ímpar}} x[n] W_N^{kn}$$

Realizando uma simples substituição de variáveis:

n par: $n = 2r$

n ímpar: $n = 2r + 1$

para $r = 0, 1, 2, \dots, N/2 - 1$, podemos escrever:

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r] W_N^{k2r} + \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r+1] W_N^{k(2r+1)}$$

o que nos leva a:

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r] (W_N^2)^{kr} + \frac{1}{\sqrt{N}} W_N^k \sum_{r=0}^{N/2-1} x[2r+1] (W_N^2)^{kr}$$

Mas note que:

$$W_N^2 = e^{-j2\pi 2/N} = e^{-j2\pi/N/2} = W_{N/2}$$

Então, temos:

$$X[k] = \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r] W_{N/2}^{kr} + W_N^k \frac{1}{\sqrt{N}} \sum_{r=0}^{N/2-1} x[2r+1] W_{N/2}^{kr}$$

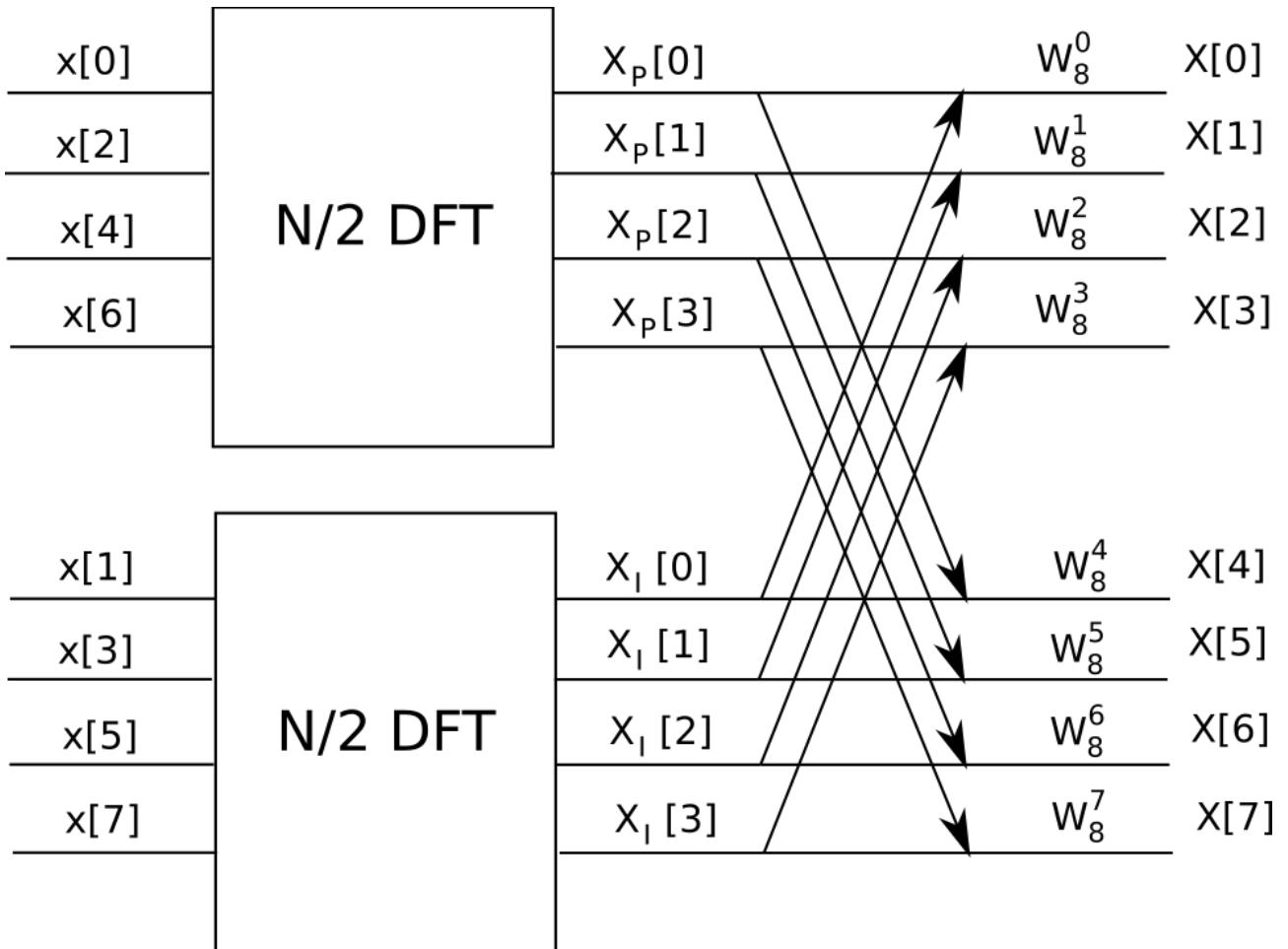
Note que o primeiro termo é justamente a DFT do sinal formado apenas pelas amostras localizadas nos índices pares do sinal, denotada por $X_p[k]$, e o segundo termo é justamente a DFT do sinal formado apenas pelas amostras localizadas nos índices ímpares do sinal, denotada por $X_I[k]$. Portanto, podemos escrever:

$$X[k] = X_p[k] + W_N^k X_I[k]$$

Na notação matemática temos:

$$\begin{aligned}
 X[0] &= X_P[0] + W_8^0 X_I[0] \\
 X[1] &= X_P[1] + W_8^1 X_I[1] \\
 X[2] &= X_P[2] + W_8^2 X_I[2] \\
 X[3] &= X_P[3] + W_8^3 X_I[3] \\
 X[4] &= X_P[0] + W_8^4 X_I[0] \\
 X[5] &= X_P[1] + W_8^5 X_I[1] \\
 X[6] &= X_P[2] + W_8^6 X_I[2] \\
 X[7] &= X_P[3] + W_8^7 X_I[3]
 \end{aligned}$$

Podemos visualizar a FFT como um diagrama de blocos. Suponha um sinal $x[]$ com $N = 8$ amostras.

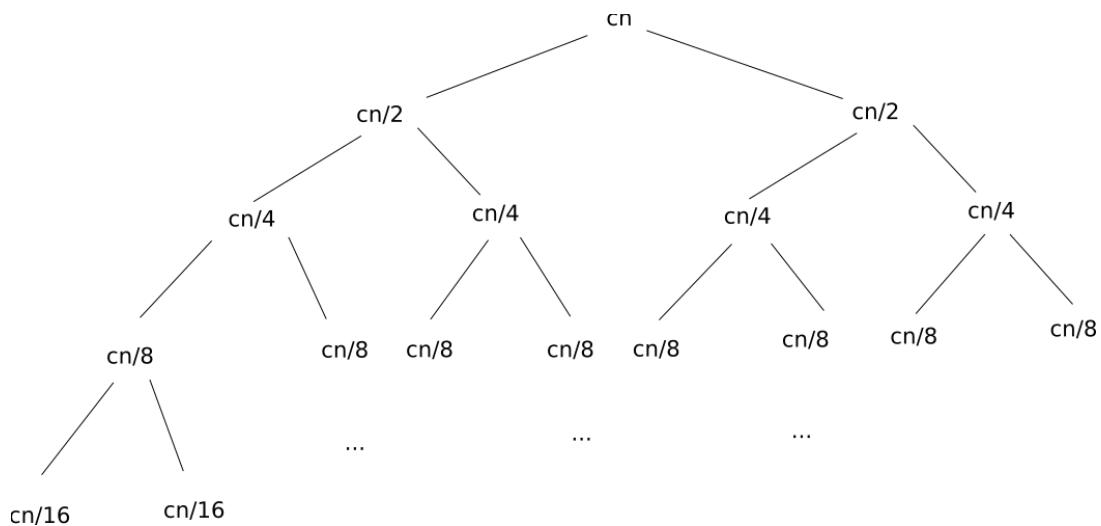


Análise da complexidade

Note que a recorrência para a FFT pode ser expressa como:

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

pois cada FFT de N pontos é decomposta em duas FFT's de $N/2$ pontos além de mais N operações multiplicações. Podemos expandir a árvore de recursão.



Sabemos que o nível máximo da árvore ocorre quando $\frac{N}{2^k}=1$, ou seja o número de níveis da árvore é $k=\log_2 N$. Somando os custos em cada nível da árvore, temos o custo total do algoritmo FFT, que é dado por:

$$T(n)=\sum_{i=0}^{\log_2 n} 2^i \frac{cn}{2^i} = \sum_{i=0}^{\log_2 n} cn = cn \log_2 n$$

o que é igual a $O(n \log_2 n)$. Para se ter uma ideia de como a FFT é mais eficiente do que a DFT, apresentamos um pequeno quadro comparativo.

Número de instruções

N	10^6	10^9
N^2 (DFT)	10^{12}	10^{18}
$N \log N$ (FFT)	20×10^6	30×10^9

Considerando que cada instrução leva em torno de 1 nanosegundo (10^{-9} segundos), no caso do sinal ter 10^9 amostras, 10^{18} nanosegundos, que seria o tempo da DFT, equivale a aproximadamente 31 anos, enquanto que 30×10^9 nanosegundos, que é o tempo da FFT, equivale a apenas 30 segundos! Veja como temos uma diferença brutal entre a complexidade quadrática e log-linear do ponto de vista assintótico.

A seguir apresentaremos e demonstraremos o Teorema Mestre, que nos permite resolver recorrências do tipo dividir para conquistar de maneira direta. Uma recorrência do tipo dividir para conquistar tem a seguinte forma:

$$T(n)=a T\left(\frac{n}{b}\right)+g(n)$$

ou seja, 1 problema de tamanho n se divide em a problemas de tamanho n/b e gera um custo extra adicional de tamanho $g(n)$.

O Teorema Mestre

Seja uma recorrência dada por:

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d$$

Então, se $n=b^k$ (potência de b, para dividir em um número exato de subproblemas) e para $k > 1$, $a \geq 1$, $b > 1$, $c > 0$ e $d \geq 0$, temos que:

1. $T(n)=O(n^d)$, se $a < b^d$
2. $T(n)=O(n^d \log n)$, se $a = b^d$
3. $T(n)=O(n^{\log_b a})$, se $a > b^d$

A seguir iremos demonstrar esse resultado teórico fundamental na análise de algoritmos. Temos que

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d$$

Aplicando a recorrência novamente, temos:

$$T(n) = a \left[aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d \right] + cn^d$$

o que nos leva a:

$$T(n) = a^2 T\left(\frac{n}{b^3}\right) + ac\left(\frac{n}{b^2}\right)^d + cn^d$$

Repetindo o processo mais uma vez, temos:

$$T(n) = a^2 \left[aT\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^d \right] + ac\left(\frac{n}{b}\right)^d + cn^d$$

o que nos leva a:

$$T(n) = a^3 T\left(\frac{n}{b^3}\right) + a^2 c\left(\frac{n}{b^2}\right)^d + ac\left(\frac{n}{b}\right)^d + cn^d$$

Seguindo com esse processo até o k-ésimo termo, chega-se na forma geral, dada por:

$$T(n) = a^k T\left(\frac{n}{b^k}\right) + a^{k-1} c\left(\frac{n}{b^{k-1}}\right)^d + a^{k-2} c\left(\frac{n}{b^{k-2}}\right)^d + \dots + ac\left(\frac{n}{b}\right)^d + cn^d$$

Mas lembre-se que $n=b^k$ (n é potência de b), então $\frac{n}{b^k}=1$, o que nos leva a:

$$T(n) = a^k T(1) + \sum_{i=0}^{k-1} a^i c\left(\frac{n}{b^i}\right)^d$$

Podemos simplificar ainda mais a equação acima tirando as constantes fora do somatório:

$$T(n) = a^k T(1) + c n^d \sum_{i=0}^{k-1} \left(\frac{a}{b^d} \right)^i \quad (*)$$

Iremos agora iniciar o caso mais simples que é o caso 2. $a=b^d$. Note que neste caso o somatório é uma constante:

$$\sum_{i=0}^{k-1} \left(\frac{b^d}{b^d} \right)^i = \sum_{i=0}^{k-1} 1^i = k$$

o que nos leva a:

$$T(n) = a^k T(1) + c n^d k$$

Mas como $n=b^k$, então $k=\log_b n$. Assim, temos:

$$T(n) = a^k T(1) + c n^d \log_b n$$

Mas, neste caso, $a=b^d$:

$$T(n) = b^{dk} T(1) + c n^d \log_b n$$

Como $b^k=n$, chega-se em:

$$T(n) = n^d T(1) + c n^d \log_b n$$

e como o segundo termo é dominante, temos que $T(n)$ é $O(n^d \log_b n)$. Para analisar os demais casos (1 e 3), primeiramente note que o somatório em (*) é de fato uma soma de uma P.G.

$$S_k = \sum_{k=0}^{k-1} \left(\frac{a}{b^d} \right)^i$$

é a soma dos k primeiros termos de uma P.G. em que o primeiro termo $a_1 = 1$ e a razão $q = \left(\frac{a}{b^d} \right)$:

$$S_k = 1 \frac{\left[\left(\frac{a}{b^d} \right)^k - 1 \right]}{\left(\frac{a}{b^d} \right) - 1}$$

de modo que:

$$T(n) = a^k T(1) + c n^d \frac{\left[\left(\frac{a}{b^d} \right)^k - 1 \right]}{\left(\frac{a}{b^d} \right) - 1}$$

Multiplicando o numerador e o denominador por b^d :

$$T(n) = a^k T(1) + c n^d \left[\frac{\frac{a^k}{(b^d)^{k-1}} - b^d}{a - b^d} \right]$$

Aplicando a distributiva:

$$T(n) = a^k T(1) + c \left[\frac{n^d a^k}{(b^d)^{k-1}} \right] - c n^d \left(\frac{b^d}{a - b^d} \right)$$

Mas lembre que:

$$\frac{n^d}{(b^d)^{k-1}} = \frac{(b^k)^d}{(b^d)^{k-1}} = \frac{b^{kd}}{b^{kd-d}} = \frac{b^{kd}}{b^{kd} b^{-d}} = b^d$$

Isso nos permite escrever:

$$T(n) = a^k T(1) + \frac{c a^k b^d}{a - b^d} - \frac{c n^d b^d}{a - b^d}$$

Agrupando os termos iniciais em a^k (colocando em evidência):

$$T(n) = a^k \left(T(1) + \frac{c b^d}{a - b^d} \right) - n^d \left(\frac{c b^d}{a - b^d} \right)$$

Lembre-se que $n = b^k$, então $k = \log_b n$. Logo:

$$a^k = a^{\log_b n} = n^{\log_b a}$$

Assim, temos:

$$T(n) = n^{\log_b a} \left(T(1) + \frac{c b^d}{a - b^d} \right) - n^d \left(\frac{c b^d}{a - b^d} \right)$$

Podemos agora, multiplicar o numerador e o denominador do segundo termo por -1:

$$T(n) = n^{\log_b a} \left(T(1) + \frac{c b^d}{a - b^d} \right) + n^d \left(\frac{c b^d}{b^d - a} \right)$$

Iremos verificar a seguir que, na expressão anterior, se $a > b^d$ o primeiro termo domina, enquanto que se $a < b^d$ o segundo termo domina. Dessa forma, analisando o primeiro caso, temos:

Caso 1. $a < b^d$

$$\log_b a < \log_b b^d$$

$$\log_b a < d$$

$$n^{\log_b a} < n^d$$

Isso significa que o segundo termo de $T(n)$ domina o primeiro!

Caso 3. $a > b^d$

$$\log_b a > \log_b b^d$$

$$\log_b a > d$$

$$n^{\log_b a} > n^d$$

Isso significa que o primeiro termo de $T(n)$ domina o segundo!

Portanto, temos que:

$$T(n) = O(n^d) \text{ , se } a < b^d \text{ e}$$

$$T(n) = O(n^{\log_b a}) \text{ , se } a > b^d$$

Vejamos a seguir um simples exemplo de aplicação do Teorema Mestre.

Ex: Seja A um algoritmo com a seguinte relação de recorrência:

$$T(n) = 2T\left(\frac{n}{3}\right) + 5$$

Qual é a complexidade de A? Aplicando o Teorema Mestre, temos que:

$$a = 2, b = 3, c = 1, d = 0$$

Como $b^d = 3^0 = 1$, temos que $a > b^d$. Portanto, $T(n) = O(n_3^{\log 2}) = O(n^{0.63})$.

"If you're always trying to be normal you will never know how amazing you can be."
-- Maya Angelou

Programação Dinâmica

A ideia da programação dinâmica consiste em resolver problemas combinando soluções de subproblemas. Porém, diferentemente da estratégia dividir para conquistar, onde os subproblemas são disjuntos, na programação dinâmica os problemas possuem sobreposições, ou seja, problemas maiores compartilham problemas menores.

Estratégia: resolver subproblema uma única vez (a primeira vez que aparece) armazenando resultado em uma tabela, de modo a evitar retrabalho e aumento do custo computacional.

Trata-se de uma estratégia muito útil para problemas de otimização.

Para desenvolver um algoritmo usando programação dinâmica, devemos:

1. Caracterizar a estrutura de uma solução ótima.
2. Recursivamente, definir o valor de uma solução ótima.
3. Computar o valor de uma solução ótima (tipicamente de maneira bottom-up).
4. Construir uma solução ótima a partir da informação computada.

Iniciaremos com o estudo de um problema clássico da computação: a sequência de Fibonacci.

A série de Fibonacci

A série de Fibonacci é um exemplo clássico de recursão, pois:

$$\begin{aligned} F(0) &= 0 && \text{(caso base 1)} \\ F(1) &= 1 && \text{(caso base 2)} \end{aligned}$$

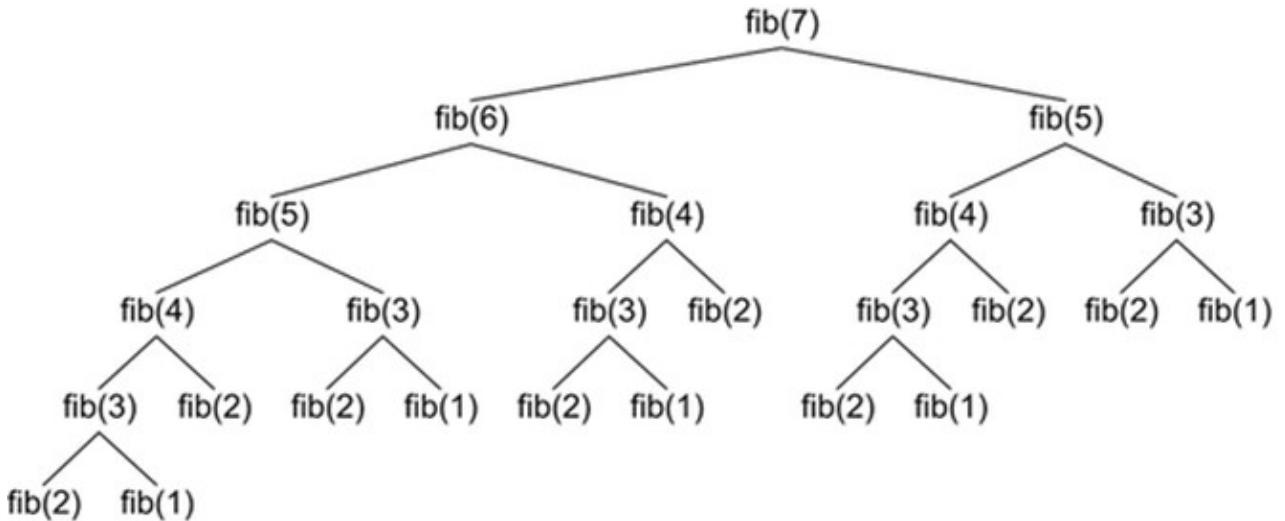
$$F(n) = F(n - 1) + F(n - 2), \text{ para todo } n > 1$$

Essa lei de formação gera a seguinte sequência: 0, 1, 1, 2, 3, 4, 8, 13, 21, 34, 55, ...

A função recursiva a seguir gera o n -ésimo termo da sequência.

```
Fib(n) {
    if n == 0
        return 0
    else {
        if n == 1 or n == 2
            return 1
        else
            return Fib(n - 1) + Fib(n - 2)
    }
}
```

Problema com essa função recursiva: muito ineficiente. Suponha que deseja-se calcular o valor de $Fib(7)$. Não é muito eficiente de ponto de vista computacional. Isso ocorre pois durante o cálculo de $F(n)$, os valores de $F(n-2)$, $F(n-3)$, etc... são calculados várias vezes. O número de chamadas recursivas cresce exponencialmente. O padrão recursivo consiste na expansão de uma árvore binária.



Note que subproblemas possuem sobreposição: aplicar programação dinâmica!

Antes de prosseguir, vamos calcular a complexidade dessa função recursiva. Note que podemos definir a seguinte recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Como para n grande, temos que $T(n-1) \approx T(n-2)$, iremos obter uma aproximação.

$$T(n) = 2T(n-1) + O(1)$$

$$T(n) = 2(2T(n-2) + O(1)) + O(1) = 2^2T(n-2) + O(1)$$

$$T(n) = 2(2(2T(n-3) + O(1)) + O(1)) + O(1) = 2^3T(n-3) + O(1)$$

Prosseguindo com essas substituições até o k -ésimo termo, chega-se em:

$$T(n) = 2^k T(n-k) + O(1)$$

A condição de parada da recursão é termos $T(0)$, ou seja, $k = n$. Logo, temos:

$$T(n) = 2^n T(0) + O(1) = 2^n + O(1)$$

Portanto, $T(n) = O(2^n)$, mostrando que a função possui complexidade exponencial (proibitiva).

A programação dinâmica, em geral, pode ser implementada a partir de 2 estratégias distintas:

1. Top-Down: através de memorização (tabela auxiliar para evitar recálculo dos subproblemas)
2. Bottom-Up: através da reversão do problema (abordagem não recursiva)

Veremos agora como utilizar a memorização para melhorar a função Fibonacci.

```

Fib_M(n) {
    m[0..n] = 0
    if n == 1 or n == 2
        m[n] = 1
    else {
        if m[n] == 0
            m[n] = Fib(n - 1) + Fib(n - 2)
    }
    return m[n]
}

```

Note que a função Fib_M calcula cada subproblema apenas uma única vez, ou seja, diminui a complexidade significativamente, pois emprega uma tabela auxiliar para armazenar o resultado dos subproblemas.

Na estratégia Bottom-Up, devemos reverter a direção que o algoritmo funciona, ou seja, iniciar do caso base e progredir em direção a solução. Essa é uma outra forma de implementar a programação dinâmica. Em geral, com essa estratégia, abordamos o problema de forma iterativa e não recursiva. A função a seguir ilustra essa ideia.

```

Fib_B(n) {
    if n == 0
        return 0
    else {
        previous = 0
        current = 1
        for i = 1 to n {
            new = previous + current
            previous = current
            current = new
        }
    }
    return current
}

```

Note que a complexidade da função anterior é $O(n)$, o que é bem melhor que exponencial.

Não são todos os algoritmos recursivos que aceitam a programação dinâmica. Por exemplo, os algoritmos de ordenação MergeSort e QuickSort não podem utilizar programação dinâmica pois os subproblemas não possuem sobreposição (são totalmente disjuntos).

A seguir veremos outras aplicações da programação dinâmica como o problema da sequência de cédulas, o problema do robô coleto, o problema do corte da haste e o problemas de caminhos mínimos em grafos.

O problema da sequência de cédulas

Suponha uma sequência de n cédulas de dinheiro cujos valores são inteiros positivos c_1, c_2, \dots, c_n não necessariamente distintos. O objetivo consiste em coletar a maior quantidade de dinheiro sujeito a restrição de que duas cédulas vizinhas não podem ser coletadas.

Seja $F(n)$ o valor máximo que pode ser coletado no arranjo de tamanho n . Para derivar uma recorrência utilizando programação dinâmica iremos separar em duas possibilidades:

1. a que inclui a cédula corrente;
2. a que não inclui a cédula corrente.

Note que o máximo valor que pode ser coletado do primeiro grupo é: $c_n + F(n-2)$

que é o valor da cédula atual mais o máximo valor coletado das primeiras $n - 2$ cédulas (aqui não consideramos a cédula imediatamente anterior pois seria uma violação das regras).. O máximo valor que pode ser coletado do segundo grupo é: $F(n-1)$, que é o máximo valor coletado das $n - 1$ cédulas anteriores (aqui não considera a atual, o valor coletado até a posição anterior pode ser considerado). Sendo assim, podemos definir a seguinte recursão:

$$F(n) = \max \{ c_n + F(n-2), F(n-1) \} \quad \text{se } n > 1$$

com as seguintes condições iniciais:

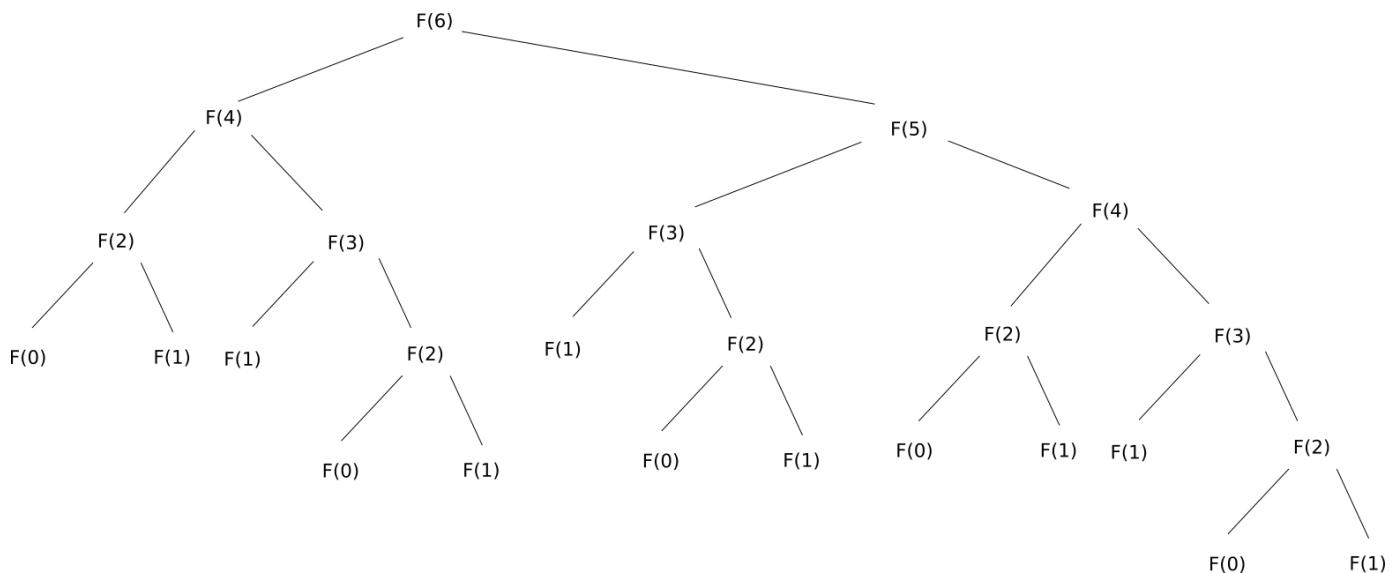
$$F(0) = 0 \quad \text{e} \quad F(1) = c_1$$

Podemos implementar a seguinte função recursiva para solucionar o problema.

```
// c é o vetor com os valores das cédulas (inicia em 1) – var. global
c = [x, y, ..., z]
```

```
F(n) {
    if n == 0
        return 0
    else {
        if n == 1
            return c[1]
        else
            return max(c[n] + F(n - 2), F(n - 1))
    }
}
```

Porém, qual é a complexidade dessa função? Vejamos a árvore de recursão. Suponha que $n = 6$, ou seja, temos apenas 6 cédulas.



Note que temos uma recorrência muito similar com a sequência de Fibonacci, pois ao invés de somar, tomamos o máximo entre dois valores, o que também é feito em $O(1)$. Podemos definir a seguinte recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Como para n grande, temos que $T(n-1) \approx T(n-2)$, iremos obter uma aproximação.

$$T(n) = 2T(n-1) + O(1)$$

Vimos que a solução de tal recorrência nos leva a conclusão de que $T(n) = O(2^n)$, mostrando que a função possui complexidade exponencial (proibitiva). Como podemos melhorar? Utilizando uma abordagem Bottom Up baseada em programação dinâmica.

```
// c é o vetor com os valores das cédulas (inicia em 1) - var. global
c = [x, y, ..., z]

Cedulas(n) {
    Define F[] as an array of size n
    F[0] = 0
    F[1] = c[1]
    for i = 2 to n
        F[i] = max(c[i]+F[i-2], F[i-1])
    return F[n]
}
```

É fácil perceber que a complexidade desse algoritmo é $O(n)$, o que é muito melhor que exponencial!

O problema do robô coleto de moedas

Suponha que diversas moedas são posicionadas em células de um tabuleiro $n \times m$, sendo apenas 1 moeda por célula. Um robô localizado no canto superior esquerdo precisa coletar o maior número de moedas possível e trazê-las para o canto inferior direito. Em cada passo, o robô pode mover uma célula para a direita ou uma célula para baixo de sua posição atual. Ao visitar uma célula com uma moeda, o robô sempre coleta a moeda em questão. Uma abordagem baseada em programação dinâmica para solucionar esse problema é descrita a seguir.

Seja $F(i, j)$ o maior número de moedas que o robô pode coletar e trazer para a célula (i, j) . Ele pode atingir essa célula tanto da célula de cima $(i-1, j)$ quanto da célula a esquerda $(i, j-1)$. Os maiores números de moedas que podem ser trazidos até essas células são $F(i-1, j)$ e $F(i, j-1)$, respectivamente.

Obviamente, não há células vizinhas acima da primeira linha, nem a esquerda da primeira coluna (nas bordas). Para as células das bordas do tabuleiro, assumimos que $F(i-1, j)$ e $F(i, j-1)$ são nulos (condições iniciais).

Assim, o maior número de moedas que podem ser trazidas até a célula (i, j) é o máximo entre esses 2 números mais uma possível moeda na célula (i, j) . Em outras palavras:

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{para } 1 \leq i \leq n, 1 \leq j \leq m$$

sujeito a

$$F(0, j) = 0 \quad \text{para } 1 \leq j \leq m$$

$$F(i, 0) = 0 \quad \text{para } 1 \leq i \leq n$$

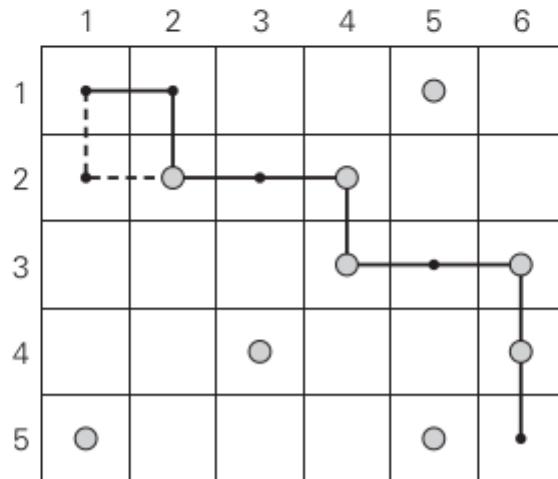
As condições iniciais representam bordas na vertical da esquerda e também na horizontal superior. Da mesma forma que os problemas anteriores, somos tentados a pensar recursivamente. Porém, neste problema, a solução recursiva se mostra inviável, pelo mesmo motivo que os anteriores (a árvore de recursão gera um número de nós que cresce exponencialmente). Trata-se de um cenário perfeito para a aplicação de técnicas de programação dinâmica. Uma ilustração gráfica do problema pode ser encontrada na figura a seguir.

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

(a)

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

(b)



(c)

O algoritmo a seguir fornece uma solução eficiente baseada na programação dinâmica para o problema em questão.

```

// C é uma matriz binária de dimensões n x m em que um valor 1
// indica presença de moeda na célula e um valor 0 indica ausência.

RoboColetor(C) {
    Define F as a n x m matrix
    F[1, 1] = C[1, 1]
    // Inicializa a primeira linha de F
    for j = 2 to m
        F[1, j] = F[1, j-1] + C[1, j]
    // Loop principal
    for i = 2 to n {
        F[i, 1] = F[i-1, 1] + C[i, 1]      // inicializa primeira coluna
        for j = 2 to n
            F[i, j] = max(F[i-1, j], F[i, j-1]) + C[i, j]
    }
    return F[n, m]
}

```

É fácil ver que a complexidade do algoritmo em questão é $O(nm)$, o que é eficiente se compararmos com uma complexidade exponencial. Para construir o caminho que o robô deve percorrer, podemos utilizar backtracking na matriz F gerada pelo algoritmo. Iniciando do canto inferior direito, temos que:

- a) se $F[i-1, j] > F[i, j-1]$ o caminho ótimo deve acessar a posição (i, j) vindo da célula superior.
- b) se $F[i-1, j] < F[i, j-1]$ o caminho ótimo deve acessar a posição (i, j) vindo da célula a esquerda.
- c) se $F[i-1, j] = F[i, j-1]$ o caminho ótimo pode vir tanto da célula superior quanto da célula a esquerda.

Com essa estratégia, note que qualquer caminho construído irá passar obrigatoriamente por $n + m$ células (pois devemos sair do canto inferior direito – destino – e voltar para o canto superior esquerdo – origem). Sendo assim, o custo computacional para a construção do caminho ótimo é $O(n+m)$.

O problema do corte da haste (The rod cutting problem)

Objetivo: dada uma haste de comprimento n e o preço associado a cada pedaço da haste, cortar e vender os pedaços de modo a maximizar o ganho.



Note que uma partição válida da haste deve satisfazer:

$$n = i_1 + i_2 + i_3 + \dots + i_k$$

onde i_k é o tamanho do k-ésimo pedaço. Supondo que $n = 4$, ou seja, a haste pode ser cortada em até 4 pedaços, todas as partições possíveis da haste são:

- a) 4
- b) 3, 1
- c) 2, 2
- d) 2, 1, 1
- e) 1, 3
- f) 1, 2, 1
- g) 1, 1, 2
- h) 1, 1, 1, 1

Isso resulta num total de $2^{n-1} = 2^3 = 8$ maneiras distintas. Iremos definir algumas variáveis:

p_i : preço de um pedaço da haste de tamanho i ($i = 1, 2, 3, \dots, n$)

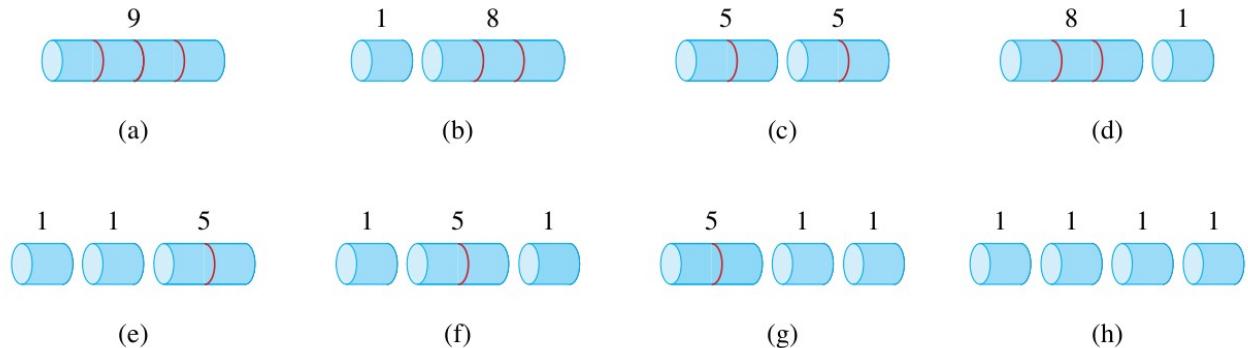
Suponha que para o nosso problema, o custo seja definido pela seguinte tabela:

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

Desejamos maximizar o valor de venda (ganho):

$$r_n = p_{i_1} + p_{i_2} + p_{i_3} + \dots + p_{i_k}$$

onde p_{i_k} é o preço do pedaço i_k . A figura a seguir ilustra os cortes e os preços dos pedaços.



Note que no nosso problema em particular ($n = 4$), temos os seguintes valores de venda:

$$4: r_n = 9$$

$$3, 1: r_n = 8 + 1 = 9$$

$$2, 2: r_n = 5 + 5 = 10$$

$$2, 1, 1: r_n = 5 + 1 + 1 = 7$$

$$1, 1, 1, 1: r_n = 1 + 1 + 1 + 1 = 4$$

Note que, na prática, mesmo que não precisemos avaliar as 2^{n-1} possibilidades, esse número, conhecido como função de partição, é assintoticamente de:

$$\frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$$

o que ainda é exponencial em n , ou seja, o espaço de busca é muito grande para uma simples busca exaustiva (força bruta não funciona). Para iniciar a formulação do problema, note que podemos

expressar os valores r_n , para $n \geq 1$ em termos dos valores de venda ótimos das hastes menores:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \dots, r_{n-1} + r_1\}$$

onde p_n é o preço da haste sem cortes e os demais valores são referentes aos valores de venda de cortes em dois pedaços de tamanho i e $n - i$, para $i = 1, 2, \dots, n - 1$.

Note que para resolver o problema original de tamanho n , resolvemos problemas menores. Ao fazer o primeiro corte, os 2 pedaços resultantes formam instâncias menores do mesmo problema. A solução ótima final incorpora as soluções ótimas dos problemas menores. Dizemos que o problema do corte da haste exibe subestrutura ótima: soluções ótimas para um problema incorporam soluções ótimas aos subproblemas relacionados.

Podemos simplificar a formulação do problema pensada em toda decomposição de uma haste de tamanho n como o primeiro pedaço seguida da decomposição do restante:

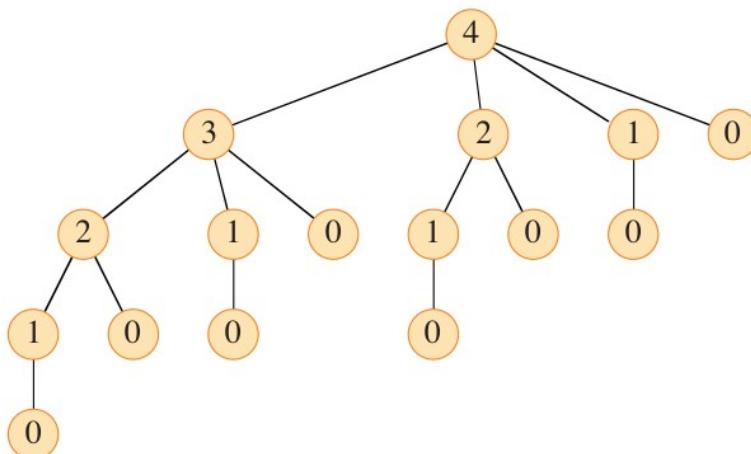
$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$$

Nessa formulação, uma solução ótima depende da solução de um subproblema e não de dois! Por isso, nos leva a um algoritmo mais eficiente.

A seguir veremos uma simples abordagem Top-Down recursiva que recebe como parâmetro um vetor de preços $p[1:n]$ e um inteiro n que representa o tamanho da haste.

```
Cut_Rod(p, n) {
    if n == 0
        return 0
    q = -inf
    for i = 1 to n
        q = max{q, p[i] + CutRod(p, n-i)}
    return q
}
```

Porém, essa implementação não é eficiente. Vejamos o que ocorre quando $n = 4$. Na primeira chamada, note que o loop FOR vai de 1 até 4. Então, ao entrar no loop temos 4 chamadas recursivas, para os valores de n iguais a 3, 2, 1 e 0. Dentro da chamada recursiva de $n = 3$, o loop FOR irá chamar a função recursivamente mais 3 vezes: 2, 1 e 0. Dessa forma, termos a seguinte árvore de recursão.



Seja $T(n)$ o número total de chamadas da função Cut_Rod (número de nós na árvore).

$$\begin{aligned}T(4) &= 15 = 2^4 - 1 \\T(3) &= 7 = 2^3 - 1 \\T(2) &= 3 = 2^2 - 1 \\T(1) &= 1 = 2^1 - 1\end{aligned}$$

Sendo assim, o custo total $T(n)$ é dado por:

$$T(n) = T(n-1) + T(n-2) + \dots + T(0) = \sum_{j=0}^{n-1} T(j) = \sum_{j=0}^{n-1} (2^j - 1) = \sum_{j=0}^{n-1} 2^j - \sum_{j=0}^{n-1} 1$$

Note que $2^{j+1} = 2 \cdot 2^j = 2^j + 2^j$, o que nos leva a $2^j = 2^{j+1} - 2^j$. Usando a soma telescópica:

$$\sum_{j=0}^{n-1} 2^j = \sum_{j=0}^{n-1} (2^{j+1} - 2^j) = 2^n - 2^0 = 2^n - 1$$

Portanto, temos que:

$$T(n) = 2^n - 1 - (n-1)$$

ou seja, a complexidade da função é $O(2^n)$ (exponencial é proibitivo).

A seguir apresentaremos uma solução Top-Down com programação dinâmica a partir da técnica de memorização.

```
Cut_Rod_Mem(p, n) {
    Let r[0:n] be an array
    for i = 0 to n
        r[i] = -inf
    return Cut_Rod_Mem_Aux(p, n, r)
}
Cut_Rod_Mem_Aux(p, n, r) {
    if r[n] >= 0
        return r[n]
    if n == 0
        q = 0
    else {
        q = -inf
        for i = 0 to n
            q = max{q, p[i] + Cut_Rod_Mem_Aux(p, n-i, r)}
    }
    r[n] = q
    return q
}
```

Note que nessa estratégia baseada em programação dinâmica a função resolve cada subproblema apenas uma única vez, ou seja, ela resolve os problemas para os tamanhos 0, 1, 2, ..., n. Mas para resolver um problema de tamanho n, o loop FOR itera por n vezes. Por essa razão, essa função tem complexidade $O(n^2)$.

Veremos a seguir uma solução Bottom-Up baseada em programação dinâmica (reversão do processo, eliminando a recursão). A ideia aqui consiste em resolver cada subproblema de tamanho j em ordem crescente.

```
Cut_Rod_B(p, n) {
    Let r[0:n] be an array
    r[0] = 0
    for j = 1 to n {
        q = -inf
        for i = 1 to j
            q = max{q, p[i] + r[j-i]}
        r[j] = q
    }
    return r[n]
}
```

A diferença aqui é que partimos de $r[0]$ e vamos subindo em direção a $r[n]$ no sentido contrário a recursão. O cálculo da complexidade do algoritmo Bottom-Up é dado pela solução do seguinte somatório:

$$T(n) = \sum_{i=1}^n i = 1 + 2 + 3 + \dots + n$$

Note que $(i+1)^2 = i^2 + 2i + 1$, o que nos leva a:

$$(i+1)^2 - i^2 = 2i + 1$$

Aplicando somatório em ambos os lados:

$$\sum_{i=1}^n [(i+1)^2 - i^2] = 2 \sum_{i=1}^n i + \sum_{i=1}^n 1$$

Pela soma telescópica, temos:

$$(n+1)^2 - 1 = 2 \sum_{i=1}^n i + n$$

Isolando o somatório finalmente nos fornece:

$$\sum_{i=1}^n i = \frac{n^2 + 2n + 1 - 1 - n}{2} = \frac{n(n+1)}{2}$$

Portanto, a complexidade dessa função é $O(n^2)$. Vimos que neste problema em particular, com o auxílio da programação dinâmica conseguimos reduzir a complexidade do algoritmo inicial de exponencial para quadrática!

Caminhos mínimos em grafos

Def: Seja $G = (V, E, w)$ um grafo ponderado com função de custo $w: E \rightarrow R^+$. O peso do caminho $P = v_0 v_1 v_2 \dots v_n$ é dado por:

$$w(P) = \sum_{i=1}^n w(v_{i-1}, v_i)$$

Define-se o caminho ótimo P^* de v_0 a v_n como aquele que minimiza $w(P)$, ou seja:

$$P^* = \underset{P}{\operatorname{argmin}} w(P)$$

se existe um caminho de v_0 a v_n . Se não existe caminho P de v_0 a v_n , então o custo do caminho é infinito, ou seja, $w(P) = \infty$.

Veremos a seguir que caminhos ótimos possuem a propriedade de subestrutura ótima.

Teorema: Seja $G = (V, E, w)$ um grafo ponderado e $P = v_0 v_1 v_2 \dots v_n$ o caminho mínimo de v_0 a v_n . Seja ainda, para $0 \leq i < j \leq n$, $P' = v_i v_{i+1} \dots v_j$ o subcaminho de P de v_i a v_j . Então, P' é o caminho mínimo de v_i a v_j .

Prova:

1. Podemos decompor P em 3 subcaminhos:



2. Logo, o peso total do caminho P é dado por:

$$w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{jn})$$

3. Suponha que exista um caminho P'_{ij} de v_i a v_j tal que $w(P'_{ij}) < w(P_{ij})$.

4. Então, o caminho \bar{P} , definido como:

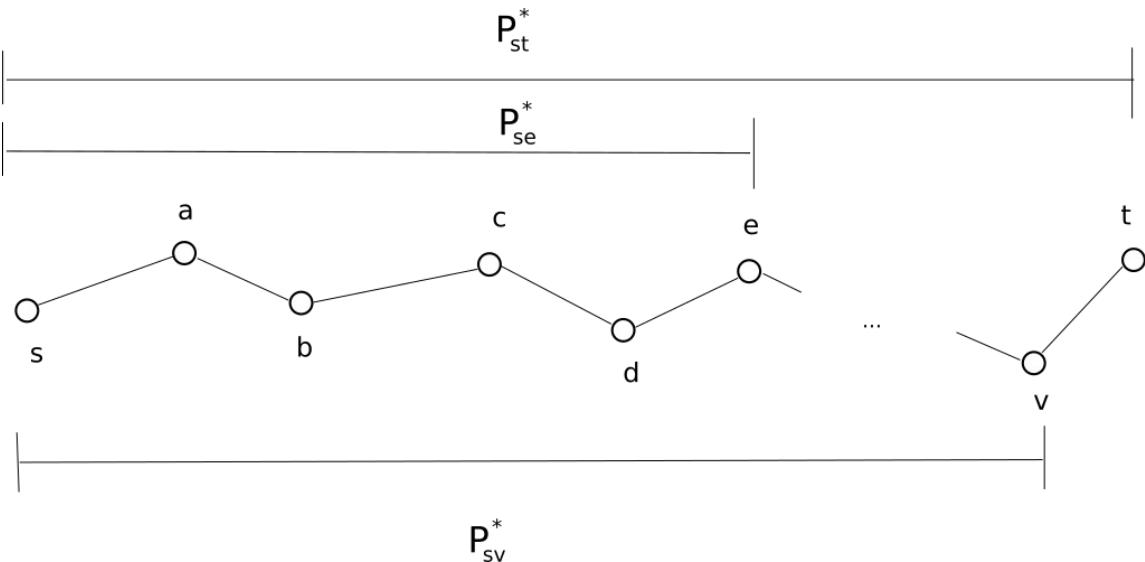


com peso:

$$w(\bar{P}) = w(P_{0i}) + w(P'_{ij}) + w(P_{jn})$$

possui $w(\bar{P}) < w(P)$, o que contradiz a hipótese de que P é o caminho mínimo. Portanto, não existe P'_{ij} diferente de P_{ij} com $w(P'_{ij}) < w(P_{ij})$.

Esse resultado é muito importante pois permite a aplicação da programação dinâmica!



Note que os caminhos mínimos P_{st}^* , P_{sv}^* e P_{se}^* compartilham arestas em comum, então temos um problema grande sendo dividido em vários subproblemas menores mas com sobreposição. A estratégia divisão e conquista não serve para esse problema, mas a programação dinâmica com certeza serve.

A seguir apresentamos o algoritmo de Dijkstra para obtenção de caminhos mínimos utilizando programação dinâmica. A ideia do algoritmo de Dijkstra é utilizar uma estratégia Bottom-Up para a reversão da recursão. Porque o algoritmo de Dijkstra é um algoritmo de programação dinâmica? Basicamente por dois motivos:

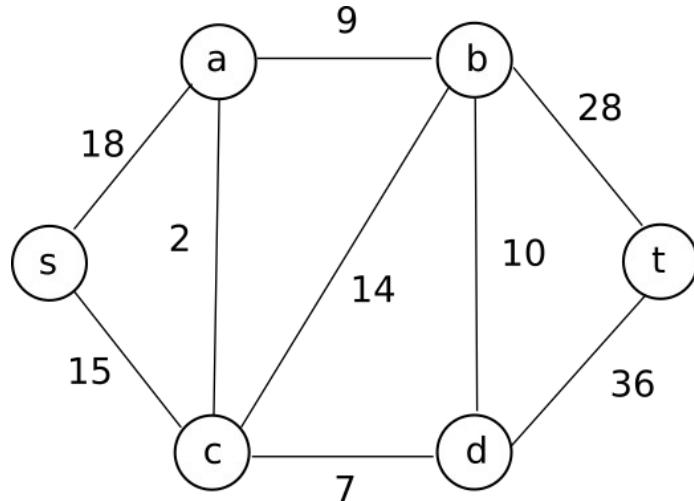
1. Não recalcula os custos dos caminhos a toda iteração: armazena os valores dos caminho mínimos em uma Fila de Prioridades Q (memorização)
2. Constrói a solução ótima a partir das soluções dos subproblemas menores (subestrutura ótima de caminhos mínimos possui sobreposição)

```

Dijkstra(G, w, s) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = nil
    }
    λ(s) = 0
    S = ∅
    Q = ∅
    for each v ∈ V
        Insert(Q, v)
    while Q ≠ ∅ {
        u = ExtractMin(Q)
        S = S ∪ {u}
        for each v in N(u) {
            λ(v) = min{λ(v), λ(u) + w(u,v)}
            if λ(v) was updated {
                π(v) = u
                Decrease_Key(Q, v, λ(v))
            }
        }
    }
}

```

A seguir ilustramos uma execução completa do algoritmo de Dijkstra em um grafo.



Fila de prioridades

	s	a	b	c	d	t
$\lambda^0(v)$	0	∞	∞	∞	∞	∞
$\lambda^1(v)$		18	∞	15	∞	∞
$\lambda^2(v)$		17	29		22	∞
$\lambda^3(v)$			26		22	∞
$\lambda^4(v)$						58
$\lambda^5(v)$						54

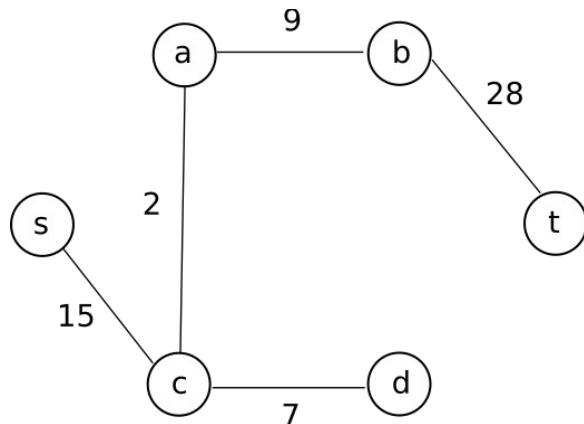
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
s	{a, c}	$\lambda(a) = \min\{\lambda(a), \lambda(s) + w(s, a)\} = \min\{\infty, 18\} = 18$ $\lambda(c) = \min\{\lambda(c), \lambda(s) + w(s, c)\} = \min\{\infty, 15\} = 15$	$\pi(a) = s$ $\pi(c) = s$
c	{a, b, d}	$\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c, a)\} = \min\{18, 17\} = 17$ $\lambda(b) = \min\{\lambda(b), \lambda(c) + w(c, b)\} = \min\{\infty, 29\} = 29$ $\lambda(d) = \min\{\lambda(d), \lambda(c) + w(c, d)\} = \min\{\infty, 22\} = 22$	$\pi(a) = c$ $\pi(b) = c$ $\pi(d) = c$
a	{b}	$\lambda(b) = \min\{\lambda(b), \lambda(a) + w(a, b)\} = \min\{29, 26\} = 26$	$\pi(b) = a$
d	{b, t}	$\lambda(b) = \min\{\lambda(b), \lambda(d) + w(d, b)\} = \min\{26, 32\} = 26$ $\lambda(t) = \min\{\lambda(t), \lambda(d) + w(d, t)\} = \min\{\infty, 58\} = 58$	---
b	{t}	$\lambda(t) = \min\{\lambda(t), \lambda(b) + w(b, t)\} = \min\{58, 54\} = 54$	$\pi(t) = b$
t	\emptyset	---	---

Mapa de predecessores

v	s	a	b	c	d	t
$\pi(v)$	---	c	a	s	c	b
$\lambda(v)$	0	17	26	15	22	54

Árvore de caminhos mínimos



Análise da complexidade

Há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo se utiliza-se estruturas de dados estáticas ou dinâmicas.

Caso 1: $G = (V, E)$ representado por uma matriz de adjacências e fila de prioridades Q representada por um array estático de n elementos (acesso direto)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(n)$ (equivale a encontrar menor elemento do vetor)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n) * O(n) + (O(1) + O(1)) * (d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1) * O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: $G = (V, E)$ representado por uma lista de adjacências e fila de prioridades Q representada por um heap binário (estruturas dinâmicas)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(\log n)$ (árvore binária)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = ExtractMin(Q)$ é $O(\log n)$ (busca em árvore binária)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (árvore binária)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(\log n) + O(n) * O(\log n) + (O(1) + O(\log n)) * (d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(\log n) + O(n \log n) + O(m \log n)$$

Como os dois últimos termos dominam os demais:

$$T(n) = O((n+m) \log n)$$

Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende!

Em grafos mais densos (m muito grande), o caso 1 é mais eficiente.

Em grafos menos densos (poucas arestas), o caso 2 é mais eficiente.

Teorema: O algoritmo de Dijkstra termina com $\lambda(v) = d(s, v)$, $\forall v \in V$, onde $d(s, v)$ é a distância geodésica de s a v (menor distância possível).

(Prova por contradição)

Obs: Note que sempre $\lambda(v) \geq d(s, v)$ (*)

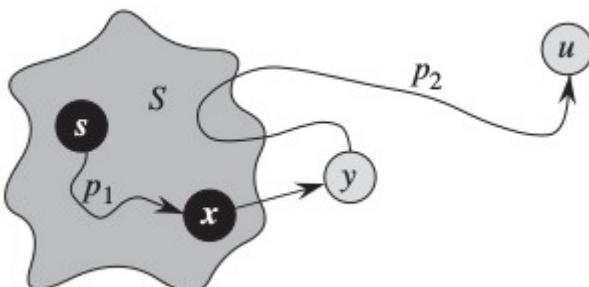
1. Suponha que u seja o 1º vértice para o qual $\lambda(u) \neq d(s, u)$ quando u entra em S.

2. Então, $u \neq s$ pois senão $\lambda(s) = d(s, s) = 0$

3. Assim, existe um caminho P_{su} pois senão $\lambda(u) = d(s, u) = \infty$. Portanto, existe um caminho mínimo P_{su}^*

4. Antes de adicionar u a S, P_{su}^* possui $s \in S$ e $u \in V - S$

5. Seja y o 1º vértice em P_{su}^* tal que $y \in V - S$ e seja x seu predecessor ($x \in S$)



$$\begin{aligned} P_{su}^* &= s \rightarrow xy \rightarrow u \\ p1 & & p2 \end{aligned}$$

Obs: Note que tanto p1 quanto p2 não precisam ter arestas

6. Como $x \in S$, $\lambda(x) = d(s, x)$ e no momento em que ele foi inserido a S, a aresta (x, y) foi relaxada, ou seja:

$$\lambda(y) = \lambda(x) + w(x, y) = d(s, x) + w(x, y) = d(s, y)$$

7. Mas y antecede a u no caminho e como $w: E \rightarrow R^+$ (pesos positivos), temos:

$$d(s, y) \leq d(s, u)$$

e portanto

$$\begin{array}{lll} \lambda(y) = d(s, y) \leq d(s, u) \leq \lambda(u) \\ (6) \quad (7) \quad (*) \end{array}$$

8. Mas como ambos y e u pertencem a $V - S$, quando u é escolhido para entrar em S temos $\lambda(u) \leq \lambda(y)$

9. Como $\lambda(y) \leq \lambda(u)$ e $\lambda(u) \leq \lambda(y)$ então temos que $\lambda(u) = \lambda(y)$, o que implica em:

$$\lambda(y) = d(s, y) = d(s, u) = \lambda(u)$$

o que gera uma contradição. Portanto $\nexists u \in V$ tal que $\lambda(u) \neq d(s, u)$ quando u entra em S.

Veremos a seguir o algoritmo Floyd-Warshall que usa programação dinâmica para obter a matriz de distâncias geodésicas ponto a ponto.

O algoritmo Floyd-Warshall

Todo caminho mínimo é composto pela origem, destino e vértices intermediários:

$$\mathbf{P} = \boxed{v_1} \ v_2 \ v_3 \ \dots \ v_{n-1} \boxed{v_n}$$

intermediários

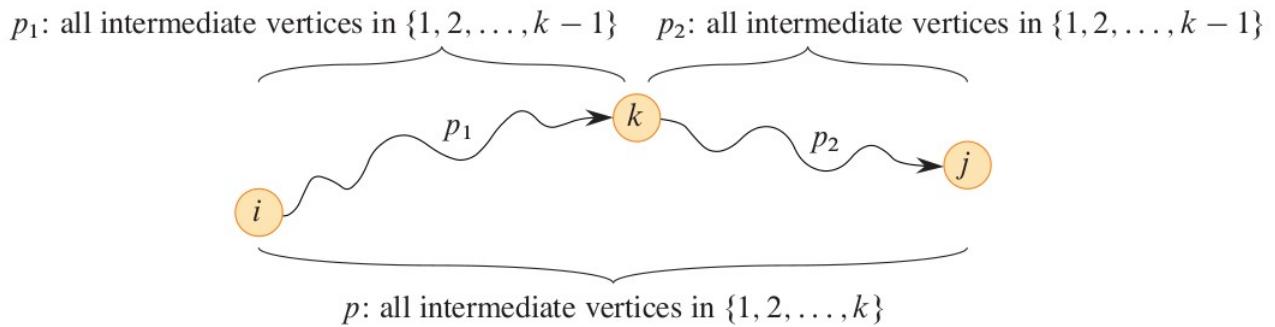
Supondo que os vértices de G são denotados por $V = \{1, 2, 3, \dots, n\}$, podemos definir o conjunto $U_k = \{1, 2, \dots, k\}$, para $1 \leq k \leq n$. Para um par de vértices arbitrário $i, j \in V$ considere todos os caminhos de i até j cujos vértices intermediários pertencem a U_k e seja P_{ij}^* o menor deles (caminho mínimo). Podemos relacionar P_{ij}^* com caminho mínimos P_{ij} de i até j que contém todos os vértices intermediários no subconjunto $U_{k-1} = \{1, 2, \dots, n-1\}$ da seguinte forma:

1. Se $k \notin P_{ij}^*$ então P_{ij} também é um caminho mínimo de i até j com todos os vértices intermediários em $U_k = \{1, 2, \dots, k\}$

2. Se $k \in P_{ij}^*$ então temos:



A figura a seguir ilustra a segunda condição.



Portanto, pela subestrutura ótima dos caminho mínimos, p_1 é o caminho mínimo de i até k com vértices intermediários em U_{k-1} e p_2 é o caminho mínimo de k até j com vértices intermediários em U_{k-1} . Ambos os caminhos p_1 e p_2 possuem vértices intermediários em U_{k-1} .

Seja $d_{ij}^{(k)}$ o custo do caminho mínimo de i até j com todos os vértices intermediários em U_k . Quando $k = 0$, não há vértices intermediários e $d_{ij}^{(0)} = w_{ij}$, onde a matriz de adjacências W é:

$$w_{ij} = \begin{cases} 0, & \text{se } i=j \\ w(v_i, v_j), & \text{se } i \neq j \text{ e } (v_i, v_j) \in E \\ \infty, & \text{se } i \neq j \text{ e } (v_i, v_j) \notin E \end{cases}$$

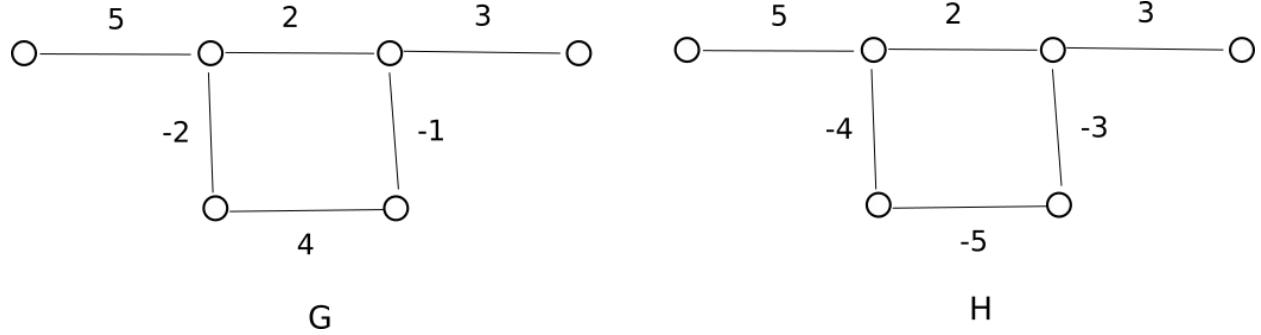
Então, com base na discussão anterior e utilizando programação dinâmica com uma abordagem Bottom-Up, podemos escrever:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & \text{se } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}, & \text{se } k \geq 1 \end{cases}$$

Como para qualquer caminho P , todos os vértices intermediários pertencem ao conjunto $\{1, 2, \dots, n\}$, a matriz $D^{(n)} = d_{ij}^{(n)}$ fornece a resposta desejada. A seguir apresentamos o algoritmo Floyd_Warshall.

```
Floyd_Warshall(W) {
    n = W.rows
    D(0) = W
    for k = 1 to n {
        Let D(k) = (d(k)ij) be a new matrix
        for i = 1 to n {
            for j = 1 to n
                d(k)ij = min{d(k-1)ij, d(k-1)ik + d(k-1)kj}
        }
    }
    return D(n)
}
```

Uma vantagem do algoritmo de Floyd-Warshall é que ele funciona mesmo na presença de arestas com pesos negativos! Porém, ele não aceita ciclos negativos. A figura a seguir ilustra a diferença entre arestas com pesos negativos e ciclos negativos.



O grafo G possui arestas negativas enquanto que o grafo H possui um ciclo negativo. Note que no grafo H quanto mais voltas dermos no ciclo em questão, menor será o caminho. Nesse caso, nem o algoritmo de Floyd-Warshall funciona corretamente.

Construindo o menor caminho

Definimos Π como sendo a matriz de predecessores. Se $k = 0$, um caminho mínimo entre i e j não possui vértices intermediários:

$$\pi_{ij}^{(0)} = \begin{cases} NIL, & \text{se } i=j \text{ ou } w_{ij} = \infty \\ i, & \text{se } i \neq j \text{ e } w_{ij} < \infty \end{cases}$$

Se $k \geq 1$, temos:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)}, & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)}, & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Note que se $d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)}$, então significa que passar pelo vértice k diminui o caminho mínimo, então devemos alterar o predecessor. A seguir mostramos um exemplo ilustrativo.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Ao final do algoritmo, suponha que desejamos saber o caminho de 2 até 5. Como fazer?

É simples: como $\pi_{25}^{(5)}=1$, sabemos que o 1 é o predecessor do 5 neste caminho.

Depois, como $\pi_{21}^{(5)}=4$, sabemos que o 4 é o predecessor do 1 neste caminho.

Pro fim, como $\pi_{24}^{(5)}=2$, sabemos que o 2 é o predecessor do 4 neste caminho.

Portanto, o caminho mínimo entre 2 e 5 é P = 2, 4, 1, 5.

Outros exemplos de problemas interessantes na computação que envolvem programação dinâmica são a multiplicação de matrizes em cadeia (Matrix-chain multiplication) e o problema da subsequência em comum mais longa (Longest common subsequence).

“The ultimate test of your knowledge is your capacity to convey it to another.”
— Professor Richard Feynman

Algoritmos Gulosos

Um algoritmo guloso sempre faz a escolha que parece ser a melhor naquele momento.

A estratégia consiste em fazer escolhas localmente ótimas na expectativa de que elas nos levem à solução ótima global.

Nem sempre algoritmos gulosos obtém soluções ótimas, mas se o problema em questão exibe subestrutura ótima, em geral, a abordagem gulosa tende a fornecer soluções ótimas.

É uma estratégia simples, mas bastante eficiente em uma variedade de problemas, como:

- Árvore geradoras mínimas (MST's)
- Código de Huffman
- Fluxo em redes (algoritmo de Ford-Fulkerson)
- Emparelhamentos estáveis (algoritmo de Gale-Shapley)
- Programação linear (algoritmo Simplex)

Algoritmos gulosos x Programação dinâmica

Apesar de parecerem similares, estratégias gulosas não são equivalentes à programação dinâmica. Para ilustrar a diferença, veremos duas variações do problema da mochila: a versão binária e a versão fracionária.

O problema da mochila binário

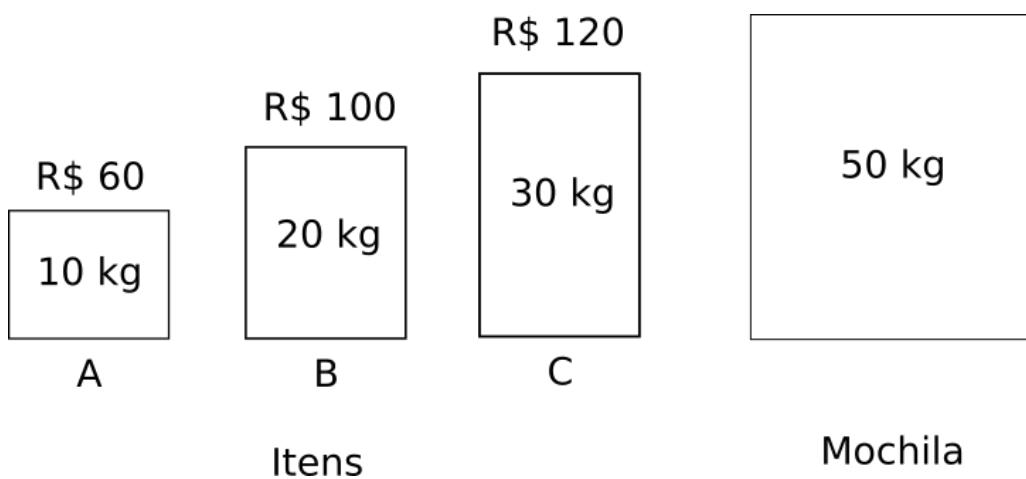
Mochila com capacidade W kg

Lista com n itens: i -ésimo item vale v_i reais e pesa w_i kg's

Deseja-se maximizar o valor sujeito a restrição de peso total W .

Na versão fracionária, é possível levar uma fração dos itens, já na binária ou leva todo objeto ou não leva nada.

Estratégia gulosa: escolher itens com maior valor por kg, ou seja, $\frac{v_i}{w_i}$



Note que temos os seguintes valores para o peso por kilo:

$$\frac{v_A}{w_A} = 6 \quad \frac{v_B}{w_B} = 5 \quad \frac{v_C}{w_C} = 4$$

- Binário: de acordo com a estratégia gulosa, devemos escolher A + B

Essa escolha resulta em $60 + 100 = 160$

Mas note que não é a escolha ótima, pois se escolhermos B + C, resultaria em $100 + 120 = 220$

- Fracionário: aqui podemos escolher A + B + 2/3 C

Essa escolha resulta em $60 + 100 + 2/3 \cdot 120 = 160 + 80 = 240$

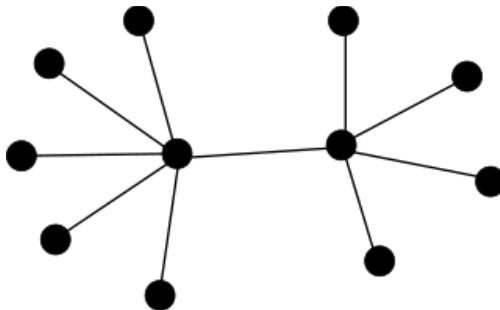
Solução ótima!

Portanto, a estratégia gulosa fornece a solução ótima apenas no caso fracionário.

Árvores geradoras mínimas

Árvores são grafos especiais com diversas propriedades únicas. Devido a essas propriedades são extremamente importantes na resolução de vários tipos de problemas práticos. Veremos ao longo do curso que vários problemas que estudaremos se resumem a: dado um grafo G, extrair uma árvore T a partir de G, de modo que T satisfaça uma certa propriedade (como por exemplo, mínima profundidade, máxima profundidade, mínimo peso, mínimos caminhos, etc).

Def: Um grafo $G = (V, E)$ é uma árvore se G é acíclico e conexo.



Teoremas e propriedades

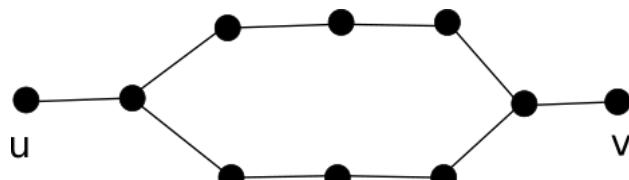
Teorema: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

(ida) $p \rightarrow q = !q \rightarrow !p$

\nexists um único caminho entre quaisquer $u, v \rightarrow G$ não é uma árvore

a) Pode existir um par u, v tal que \nexists caminho (zero caminhos). Isso implica em G desconexo, o que implica que G não é uma árvore

b) Pode existir um par u, v tal que \exists mais de um caminho.

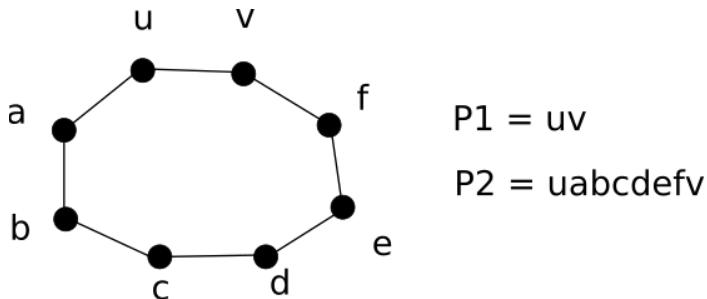


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

(volta) $q \rightarrow p = !p \rightarrow !q$

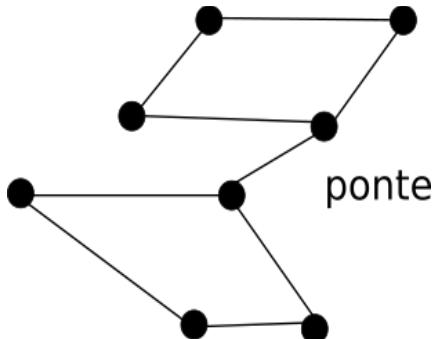
G não é árvore $\rightarrow \nexists$ único caminho entre quaisquer u, v

Para G não ser árvore, G deve ser desconexo ou conter um ciclo. Note que no primeiro caso existe um par u, v tal que não há caminho entre eles. Note que no segundo caso existem 2 caminhos entre u e v , conforme ilustra a figura



Def: Uma aresta $e \in E$ é ponte se $G - e$ é desconexo

Ou seja, a remoção de uma aresta ponte desconecta o grafo

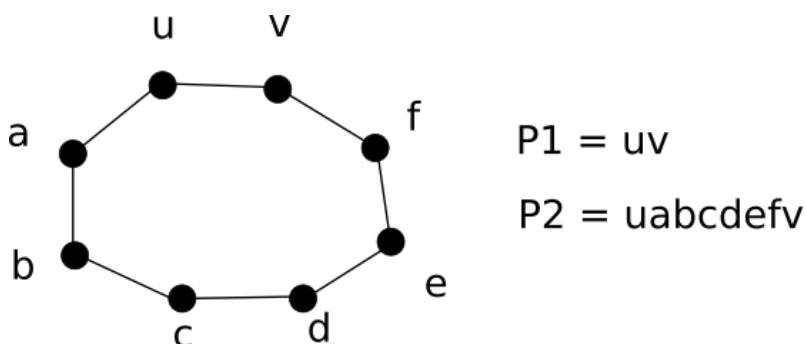


Como identificar arestas ponte?

Teorema: Uma aresta $e \in E$ é ponte \Leftrightarrow aresta não pertence a um ciclo C

(ida) $p \rightarrow q = !q \rightarrow !p$

$e \in C \rightarrow$ aresta não é ponte



Como aresta pertence a um ciclo C , há 2 caminhos entre u e v . Logo a remoção da aresta $e = (u,v)$ não impede que o grafo seja conexo, ou seja, $G - e$ ainda é conexo. Portanto, e não é ponte

(volta) $q \rightarrow p = !p \rightarrow !q$

aresta não é ponte $\rightarrow e \in C$

Se aresta não é ponte então $G - e$ ainda é conexo. Se isso ocorre, deve-se ao fato de que em $G - e$ ainda existe um caminho entre u e v que não passa por e . Logo, em G existem 2 caminhos, o que nos leva a conclusão de que a união entre os 2 caminhos gera um ciclo C .

Teorema: G é uma árvore \Leftrightarrow Toda aresta é ponte

(ida) $p \rightarrow q = !q \rightarrow !p$

\exists aresta não ponte $\rightarrow G$ não é árvore

A existência de uma aresta não ponte implica na existência de ciclo. A presença de um ciclo C faz com que G não seja um árvore

(volta) $q \rightarrow p = !p \rightarrow !q$

G não é árvore $\rightarrow \exists$ aresta não ponte

Para G não ser uma árvore, deve existir um ciclo em G . Logo, todas as arestas pertencentes ao ciclo não são pontes.

Teorema: Se $G = (V, E)$ é uma árvore com $|V| = n$ então $|E| = n - 1$

Prova por indução

$P(n)$: Toda árvore de n vértices tem $n - 1$ arestas

Base: $P(1)$: $n = 1 \rightarrow 0$ arestas (OK)

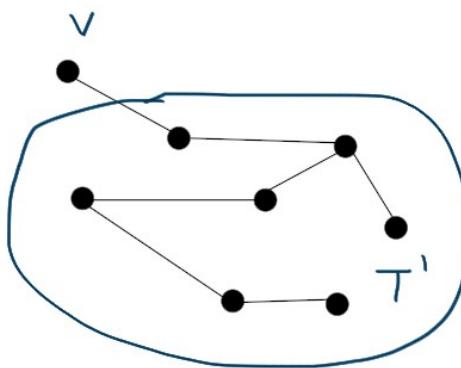
Passo de indução: $P(k) \rightarrow P(k+1)$ para k arbitrários

$P(k+1)$: Toda árvore de $n + 1$ vértices tem n arestas

1. Seja $T = (V_T, E_T)$ com $|V_T| = n + 1$

2. Seja $v \in V_T$ uma folha em T

3. Seja $T' = (V'_T, E'_T)$ a árvore obtida removendo v e a única aresta incidente a v : $T' = T - v$



4. Note que T' é uma árvore, pois como T é conexo, T' também é e como T é acíclico, T' também é. Logo, podemos dizer que $T' = (V'_T, E'_T)$ possui n' vértices e m' arestas: $|V'_T| = n'$ e $|E'_T| = m'$.
5. Mas pela hipótese de indução, toda árvore de n' vértices tem $n' - 1$ arestas, então $m' = n' - 1$
6. Como T' tem exatamente uma aresta e um vértice a menos que T :

$$m' = m - 1 = n' - 1 = (n - 1) - 1$$

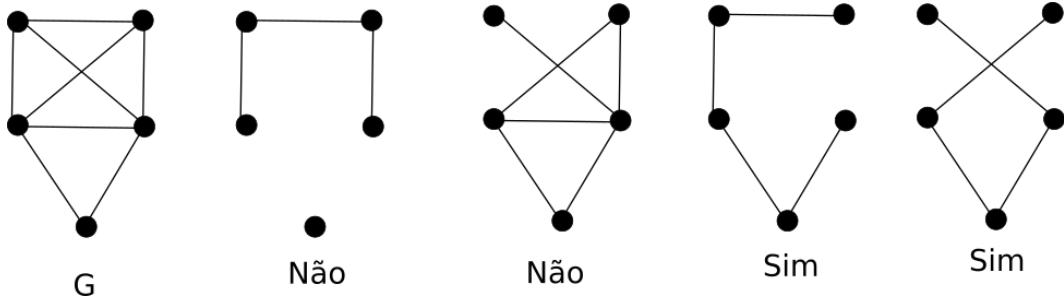
o que implica em $m - 1 = n - 1 - 1$, ou seja, $m = n - 1$. A prova está concluída.

Teorema: A soma dos graus de uma árvore de n vértices não depende da lista de graus, sendo dada por $2n - 2$

$$\sum_{i=1}^n d(v_i) = 2|E| = 2(n-1) = 2n - 2$$

Def: Árvore geradora (spanning tree)

Seja $G = (V, E)$ um grafo. Dizemos que $T = (V, E_T)$ é uma árvore geradora de G se T é um subgrafo de G que é uma árvore (ou seja tem que conectar todos os vértices)



O problema da árvore geradora mínima (MST)

Dentre todas as árvores geradoras de G , obter aquela de menor peso.

Def: Dado $G = (V, E, w)$, onde $w: E \rightarrow \mathbb{R}^+$ (peso da aresta e), obter a árvore geradora T que minimiza o seguinte critério:

$$w(T) = \sum_{e \in T} w(e) \quad (\text{soma dos pesos das arestas que compõem a árvore})$$

Por exemplo, deseja-se conectar os bairros da cidade com fibra ótica. Qual é a interligação que minimiza o custo?

Estratégia gulosa: abordagem iterativa em que a cada passo devemos escolher a aresta de menor custo que seja segura ($e \in E/T + e$ continua sendo árvore)

A seguir apresentamos uma função genérica para o problema da árvore geradora mínima.

GENERIC-MST(G, w)

- 1 $A = \emptyset$
- 2 **while** A does not form a spanning tree
- 3 find an edge (u, v) that is safe for A
- 4 $A = A \cup \{(u, v)\}$
- 5 **return** A

A pergunta que surge é: como podemos determinar se aresta é segura? Cada algoritmo tem sua própria estratégia.

O algoritmo de Kruskal

Objetivo: escolher a cada passo a aresta de menor custo que não forme um ciclo.

Ideia: iniciar adicionando cada vértice de G como raiz de uma árvore e a cada passo adicionar a aresta de menor custo com extremidades em árvores distintas.

Para isso, iremos utilizar 3 primitivas básicas:

1. Make_Set(v): cria uma árvore contendo um único vértice v (raiz)

```
Make_Set(v) {  
    v.p = v           // pai do vértice v é ele mesmo (raiz)  
    v.rank = 0        // altura da árvore (nível)  
}
```

2. Find_Set(v): retorna qual é a árvore que o vértice v pertence

```
Find_Set(v) {  
    if v != v.p          // se não é raiz  
        v.p = Find_Set(v.p) // recursão: v se torna o pai  
    return v.p            // retorna raiz da árvore de v  
}
```

3. Union(u, v): faz a fusão das raízes das árvores de u e de v, criando uma única árvore

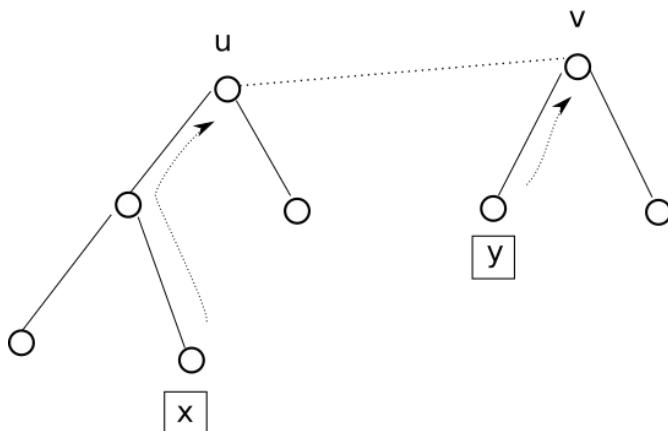
```
Union(u, v) {  
    Link(Find_Set(u), Find_Set(v))  
}
```

// Função auxiliar para fundir as árvores de u e de v

```
Link(u, v) {  
    if u.rank > v.rank  
        v.p = u  
    else {  
        u.p = v  
        if u.rank == v.rank  
            v.rank = v.rank + 1  
    }  
}
```

A figura a seguir ilustra o processo.

Union(x, y)



Ao realizar Union(x, y), primeiro encontramos as raízes das árvores de x e de y, que nesse caso são u e v respectivamente. Então, a função auxiliar Link, realiza a fusão dessas duas árvores, criando uma única. Porém, a pergunta é: quem será pai de quem? Note que nesse caso u será pai de v, pois a altura da árvore de raiz u é maior! Assim, não preciso alterar a altura dela. A maior árvore domina a menor. Somente quando as alturas das duas árvores são iguais é que precisamos incrementar a altrua (rank) em uma unidade.

A seguir apresentamos o algoritmo de Kruskal para a obtenção de uma árvore geradora mínima.

```
MST_Kruskal(G, w) {
    T = ∅
    for each v ∈ V
        Make_Set(v)
    Create a list of the edges e ∈ E
    Sort the list of edges in increasing order by weight w
    for each e = (u, v) from the sorted list {
        if Find_Set(u) ≠ Find_Set(v) {
            T = T ∪ {(u, v)}
            Union(u, v)
        }
    }
}
```

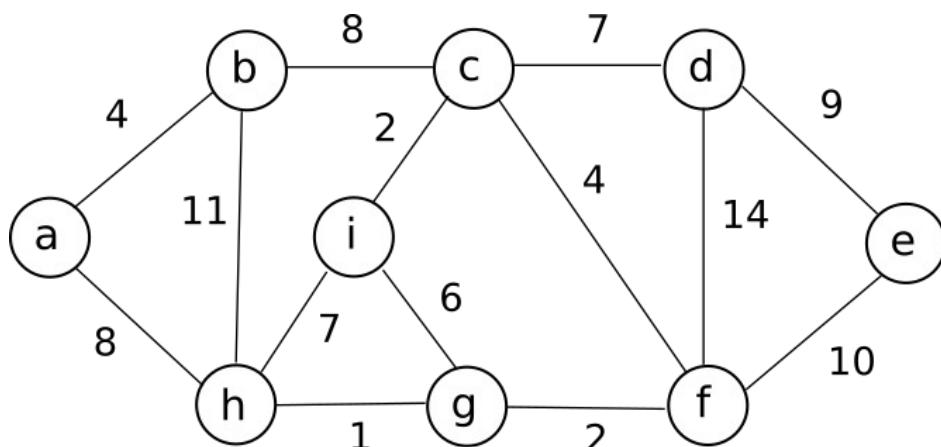
Note que trata-se de um algoritmo guloso pois ele sempre tenta a aresta de menor peso. A seguir iremos realizar um trace do algoritmo (simulação passo a passo). Para isso iremos considerar a seguinte notação:

E^- : conjunto das arestas de peso mínimo não seguras ($\text{find_set}(u) = \text{find_set}(v)$)

E^+ : conjunto das arestas de peso mínimo seguras ($\text{find_set}(u) \neq \text{find_set}(v)$)

e_k : aresta escolhida no passo k

Ex: Suponha que os vértices representem bairros e as arestas com pesos os custos de interligação desses bairros (fibra ótica)



A lista das arestas ordenadas por peso é:

$$E_w = [1, 2, 2, 4, 4, 6, 7, 7, 8, 8, 9, 10, 11, 14]$$

$$E = \{(g, h), (c, i), (f, g), (a, b), (c, f), (g, i), (c, d), (a, h), (b, c), (d, e), (e, f), (b, h), (d, f)\}$$

k	E^-	E^+	e_k
1	-	$\{(g,h)\}$	(g,h)
2	-	$\{(c,i), (f,g)\}$	(c,i)
3	-	$\{(g,f)\}$	(g,f)
4	-	$\{(a,b), (c,f)\}$	(a,b)
5	-	$\{(c,f)\}$	(c,f)
6	$\{(g,i)\}$	-	-
7	$\{(h,i)\}$	$\{(c,d)\}$	(c,d)
8	-	$\{(a,h), (b,c)\}$	(a,h)
9	$\{(b,c)\}$	-	-
10	-	$\{(d,e)\}$	(d,e)

A seguir iremos demonstrar a otimalidade do algoritmo de Kruskal.

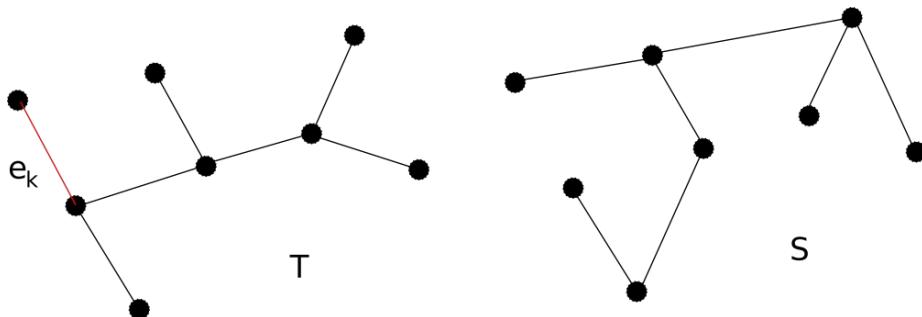
Teorema: Toda árvore T gerada pelo algoritmo de Kruskal é uma MST de G

Esse resultado garante que o algoritmo sempre funciona e é ótimo (retorna sempre a solução ótima)

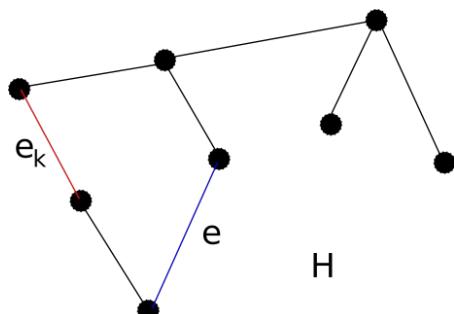
Prova por contradição

Seja T é a árvore retornada por Kruskal.

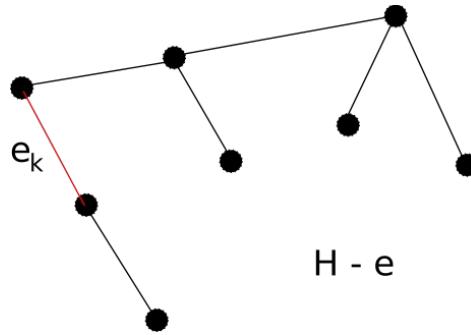
1. Supor que $\exists S \neq T$ tal que $w(S) < w(T)$ (S é uma árvore)
2. Seja $e_k \in T$ a primeira aresta adicionada em T que não está em S (pois árvores são diferentes)



3. Faça $H = (S + e_k)$. Note que H não é mais uma árvore e contém um ciclo



4. Note que no ciclo C , $\exists e \in S$ tal que $e \notin T$ (pois senão C existiria em T). O subgrafo $H - e$ é conexo, possui $n - 1$ arestas e define uma árvore geradora de G .

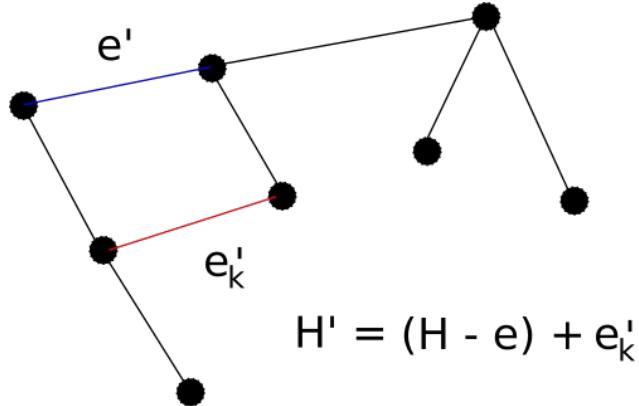


5. Porém, $w(e_k) \leq w(e)$ e assim $w(H - e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, é garantido pela ordenação)

Lista de arestas (após ordenação)



6. Repetindo o processo usado para gerar $H - e$ a partir de S é possível produzir uma sequência de árvores que se aproximam cada vez mais de T .



$$S \rightarrow (H - e) \rightarrow (H' - e') \rightarrow (H'' - e'') \rightarrow \dots \rightarrow T$$

de modo que

$$w(S) \geq w(H - e) \geq w(H' - e') \geq \dots \geq w(T) \quad (\text{contradição a suposição inicial})$$

Portanto, não existe árvore com peso menor que T , mostrando que T tem peso mínimo. (Não há como S ter peso menor que T).

Análise da complexidade

Primeiramente, devemos analisar as complexidades das primitivas básicas.

É fácil perceber que a função `Make_Set(v)` é $O(1)$. Como ela é executada n vezes, temos que no total o custo será $O(n)$.

A ordenação das arestas pode ser realizada com o MergeSort ou QuickSort, que são $O(m \log m)$, onde m denota o número de arestas.

A primitiva $\text{Find_Set}(v)$ deve retornar a raiz da árvore que v pertence. Note que para isso, dependemos da altura da árvore. No caso em que v é um nó folha da árvore binária, $h = \log n$. Portanto, a complexidade da função é $O(\log n)$. Se a árvore não for binária, muda-se apenas a base do logaritmo. Logo, a primitiva $\text{Union}(u, v)$ que faz uso de $\text{Find_Set}(v)$ também é $O(\log n)$.

Note que $\text{Find_Set}(v)$ é executada duas vezes para cada aresta (uma para cada extremidade: u e v). Assim, o custo total é $2m O(\log n)$, o que é $O(m \log n)$.

A primitiva $\text{Union}(u, v)$ é executada somente quando uma aresta é adicionada a árvore. Como uma árvore tem $m = n-1$ arestas, a complexidade é $O(n \log n)$.

Portanto, o custo total do algoritmo de Kruskal é:

$$C = O(n) + O(m \log m) + O(m \log n) + O(n \log n)$$

Podemos perceber que o termo dominante é $O(m \log m)$. Como $m < n^2$, temos que:

$$\log m < \log n^2 = 2 \log n = O(\log n)$$

ou seja, a complexidade do algoritmo de Kruskal é $O(m \log n)$.

Algoritmo de Prim

O algoritmo de Prim também utiliza uma abordagem gulosa para a construção da MST.

Ideia: iniciar de uma raiz r e a cada passo adicionar a aresta de menor custo com uma extremidade no conjunto dos vértices visitados e outra extremidade no conjunto dos vértices não visitados.

Definição das variáveis

$\lambda(v)$ ou $v.\text{key}$: menor custo de entrada para o vértice v até o momento

$\pi(v)$ ou $v.\pi$: predecessor do vértice v em T

Q : fila de prioridades (maior prioridade = menor $\lambda(v)$)

Veremos a seguir que esse algoritmo é muito similar ao algoritmo de Dijkstra.

```
MST_Prim(G, w, r) {
    for each  $v \in V$  {
         $\lambda(v) = \infty$ 
         $\pi(v) = \text{nil}$ 
    }
     $\lambda(r) = 0$ 
    // Insert the vertices in  $Q$ 
     $Q = \emptyset$ 
    for each  $v \in V$ 
        Insert( $Q$ ,  $v$ )
```

```

while Q ≠ ∅ {
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v in N(u) {
        if v ∈ Q and w(u,v) < λ(v) { // found lighter edge
            λ(v) = w(u,v)
            π(v) = u
            Decrease_Key(Q, v, w(u,v))
        }
    }
}

```

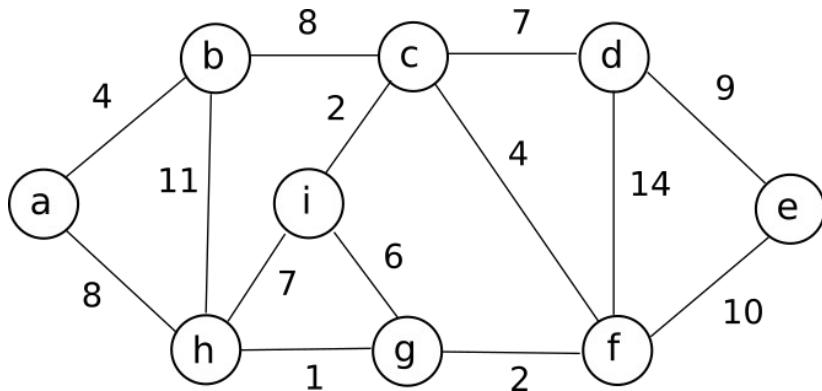
Note that the internal IF in the FOR loop is equivalent to:

```

if v ∈ Q {
    λ(v) = min{λ(v), w(u,v)}
    if λ(v) was updated {
        π(v) = u
        Decrease_Key(Q, v, w(u,v))
    }
}

```

A seguir veremos um trace da execução completa do algoritmo de Prim em um simples exemplo.



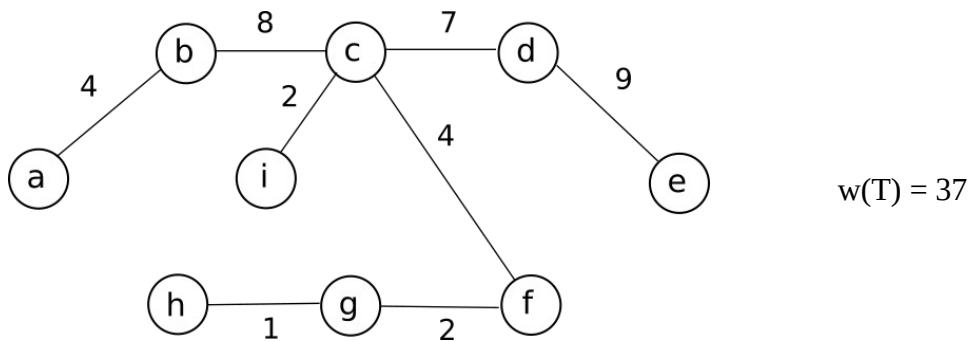
Fila de prioridades

	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞							
$\lambda^{(1)}(v)$		4	∞	∞	∞	∞	∞	8	∞
$\lambda^{(2)}(v)$			8	∞	∞	∞	∞	8	∞
$\lambda^{(3)}(v)$				7	∞	4	∞	8	2
$\lambda^{(4)}(v)$				7	∞	4	6	7	
$\lambda^{(5)}(v)$				7	10		2	7	
$\lambda^{(6)}(v)$				7	10			1	
$\lambda^{(7)}(v)$					9				

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b) = \min\{\lambda(b), w(a,b)\} = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\lambda(h), w(a,h)\} = \min\{\infty, 8\} = 8$	$\pi(b) = a$ $\pi(h) = a$
b	{c, h}	$\lambda(c) = \min\{\lambda(c), w(b,c)\} = \min\{\infty, 8\} = 8$ $\lambda(h) = \min\{\lambda(h), w(b,h)\} = \min\{8, 11\} = 8$	$\pi(c) = b$ ---
c	{d, f, i}	$\lambda(d) = \min\{\lambda(d), w(c,d)\} = \min\{\infty, 7\} = 7$ $\lambda(f) = \min\{\lambda(f), w(c,f)\} = \min\{\infty, 4\} = 4$ $\lambda(i) = \min\{\lambda(i), w(c,i)\} = \min\{\infty, 2\} = 2$	$\pi(d) = c$ $\pi(f) = c$ $\pi(i) = c$
i	{h, g}	$\lambda(h) = \min\{\lambda(h), w(i,h)\} = \min\{8, 7\} = 7$ $\lambda(g) = \min\{\lambda(g), w(i,g)\} = \min\{\infty, 6\} = 6$	$\pi(h) = i$ $\pi(g) = i$
f	{d, e, g}	$\lambda(d) = \min\{\lambda(d), w(f,d)\} = \min\{7, 14\} = 7$ $\lambda(e) = \min\{\lambda(e), w(f,e)\} = \min\{\infty, 10\} = 10$ $\lambda(g) = \min\{\lambda(g), w(f,g)\} = \min\{6, 2\} = 2$	---
g	{h}	$\lambda(h) = \min\{\lambda(h), w(g,h)\} = \min\{7, 1\} = 1$	$\pi(h) = g$
h	\emptyset	---	---
d	{e}	$\lambda(e) = \min\{\lambda(e), w(d,e)\} = \min\{10, 9\} = 9$	$\pi(e) = d$
e	\emptyset	---	---

MST

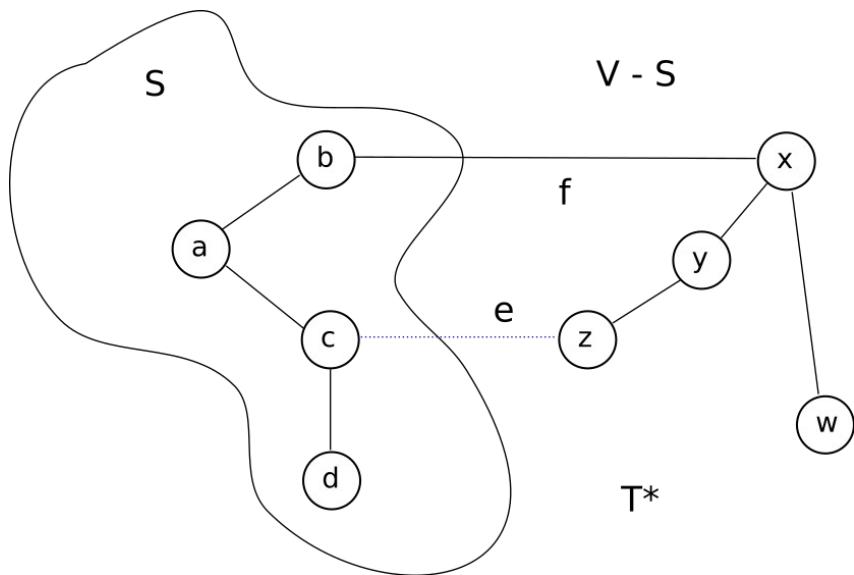


Mapa de predecessores

v	a	b	c	d	e	f	g	h	i
$\pi(v)$	--	a	b	c	d	c	f	g	c
$\lambda(v)$	0	4	8	7	9	4	2	1	2

A seguir veremos um resultado fundamental para provar a otimalidade do algoritmo de Prim.

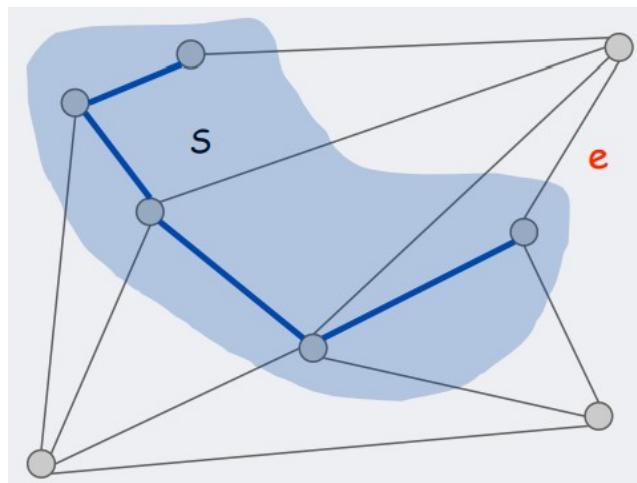
Propriedade do corte: Seja $G = (V, E, w)$ um grafo, S um subconjunto qualquer de V e $e \in E$ a aresta de menor custo com exatamente uma extremidade em S . Então, a MST de G contém e .



Prova por contradição: Seja T^* uma MST de G . Suponha que $e \notin T^*$. Ao adicionar e em T^* cria-se um único ciclo C . Para que $T^* - e$ fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (senão T^* seria desconexo). Então $T = T^* + e - f$ também é árvore geradora. Como, $w(e) < w(f)$, segue que T tem peso mínimo. Portanto, T^* não pode ser MST de G .

Teorema: A árvore T obtida pelo algoritmo de Prim é uma MST de G .

Seja S o subconjunto de vértices de G na árvore T (definido pelo algoritmo). O algoritmo de Prim adiciona em T a cada passo a aresta de menor custo com apenas um vértice extremidade em S . Portanto, pela propriedade do corte, toda aresta adicionada pertence a MST de G .



Análise da complexidade

A análise da complexidade do algoritmo de Prim é muito similar a realizada para o algoritmo de Dijkstra.

Há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo se utiliza-se estruturas de dados estáticas ou dinâmicas.

Caso 1: $G = (V, E)$ representado por uma matriz de adjacências e fila de prioridades Q representada por um array estático de n elementos (acesso direto)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = ExtractMin(Q)$ é $O(n)$ (equivale a encontrar menor elemento do vetor)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n)*O(n) + (O(1) + O(1))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1)*O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: $G = (V, E)$ representado por uma lista de adjacências e fila de prioridades Q representada por um heap binário (estruturas dinâmicas)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(\log n)$ (árvore binária)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = ExtractMin(Q)$ é $O(\log n)$ (busca em árvore binária)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (árvore binária)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(\log n) + O(n)*O(\log n) + (O(1) + O(\log n))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(\log n) + O(n \log n) + O(m \log n)$$

Como os dois últimos termos dominam os demais:

$$T(n) = O((n+m) \log n)$$

Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende!

Em grafos mais densos (m muito grande), o caso 1 é mais eficiente.

Em grafos menos densos (poucas arestas), o caso 2 é mais eficiente.

O código de Huffman

Algoritmo guloso muito utilizado para compressão de dados sem perdas (zip).

Utilizado na codificação de dados, pois é capaz de gerar códigos binários de tamanhos variáveis baseados na probabilidade de ocorrência dos símbolos.

Na codificação tradicional, todos os símbolos do alfabeto possuem códigos do mesmo tamanho.

Seja $\alpha = (a, b, c, d, e, f)$ o alfabeto de símbolos a serem codificados. Para cada símbolo $\alpha_i \in \alpha$, seja w_i a probabilidade de ocorrência de α_i nos dados. A ideia do código de Huffman é gerar códigos menores para os símbolos mais prováveis e códigos maiores para símbolos menos prováveis, fazendo com que na média, o número de bits utilizado na codificação seja menor.

Suponha que nos nossos dados, as probabilidades w_i de cada símbolo sejam dadas por:

	a	b	c	d	e	f	
w	0.45	0.13	0.12	0.16	0.09	0.05	(soma igual a 1)

Na codificação tradicional, devemos utilizar 3 bits, pois com apenas 2 conseguimos identificar apenas 4 símbolos diferentes. Então, iniciando do símbolo a, temos a seguinte codificação:

	a	b	c	d	e	f
C	000	001	010	011	100	101
L(C)	3	3	3	3	3	3

Note que o comprimento de todos os códigos é fixo e igual a 3. Isso significa que, precisamos de 3 bits por símbolo, pois:

$$l(C) = \sum_{i=1}^n w_i l_i = 0.45 \times 3 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 3 + 0.05 \times 3 = 3$$

O código de Huffman para esses símbolos pode ser calculado a partir das probabilidades w_i . Veremos mais adiante como fazer isso. Por hora, basta sabermos que os códigos gerados pelo método de Huffman são:

	a	b	c	d	e	f
C	0	101	100	111	1101	1100
L(C)	1	3	3	3	4	4

Note que neste caso, o tamanho médio do código (valor esperado) é dado por:

$$l(C) = \sum_{i=1}^n w_i l_i = 0.45 \times 1 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 4 + 0.05 \times 4 = 0.84 + 0.84 + 0.42 = 2.1$$

Isso significa que, em média o número de bits necessário para codificar um símbolo é 2.1, o que é cerca de 30% menor que a codificação tradicional!

Abordagem gulosa

A seguir definimos as variáveis

1. α : conjunto de n caracteres (alfabeto)
2. $\forall \alpha_i \in \alpha$ seja $w(\alpha_i)$ a probabilidade de ocorrência de cada símbolo
3. Q: Fila de prioridades (quanto menor $w(\alpha_i)$, maior a prioridade)

O algoritmo a seguir apresenta o código de Huffman.

```
Huffman( $\alpha$ ) {
    n =  $|\alpha|$            // number of symbols
    // Insert symbols in Q
    for each  $\alpha_i \in \alpha$ 
        Insert(Q,  $\alpha_i$ )
    for i = 1 to n-1 {
        Create a new node z
        x = ExtractMin(Q)
        y = ExtractMin(Q)
        z.left = x
        z.right = y
        w(z) = w(x) + w(y)
        Insert(Q, z)
    }
    // returns the root of the tree
    return ExtractMin(Q)
}
```

Porque algoritmo de Huffman é guloso?

Note que ele utiliza uma abordagem Bottom-Up (inicia pelas folhas). Uma vez que o algoritmo inicia removendo da fila de prioridades os 2 símbolos de menor probabilidade, eles serão os nós mais distantes da raiz e por isso terão o maior código binário.

A seguir veremos um exemplo de como gerar o código de Huffman para um conjunto de símbolos.

Ex:	a	b	c	d	e	f	(alfabeto)
w	0.45	0.13	0.12	0.16	0.09	0.05	(probabilidades)

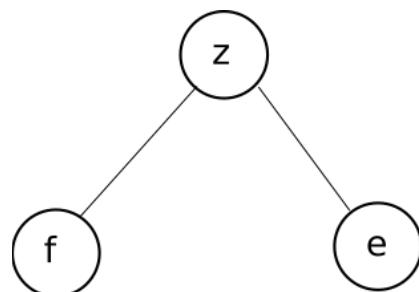
i = 1

$$Q^{(1)} = \{a:0.45, b:0.13, c:0.12, d:0.16, e:0.09, f:0.05\}$$

$$x=f$$

$$y=e$$

$$w(z) = w(x) + w(y) = 0.09 + 0.05 = 0.14$$



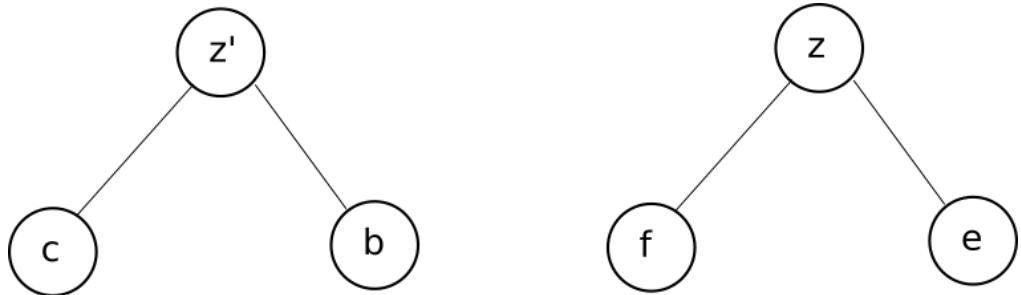
i = 2

$$Q^{(2)} = \{a:0.45, b:0.13, c:0.12, d:0.16, z:0.14\}$$

$$x=c$$

$$y=b$$

$$w(z') = w(x) + w(y) = 0.12 + 0.13 = 0.25$$



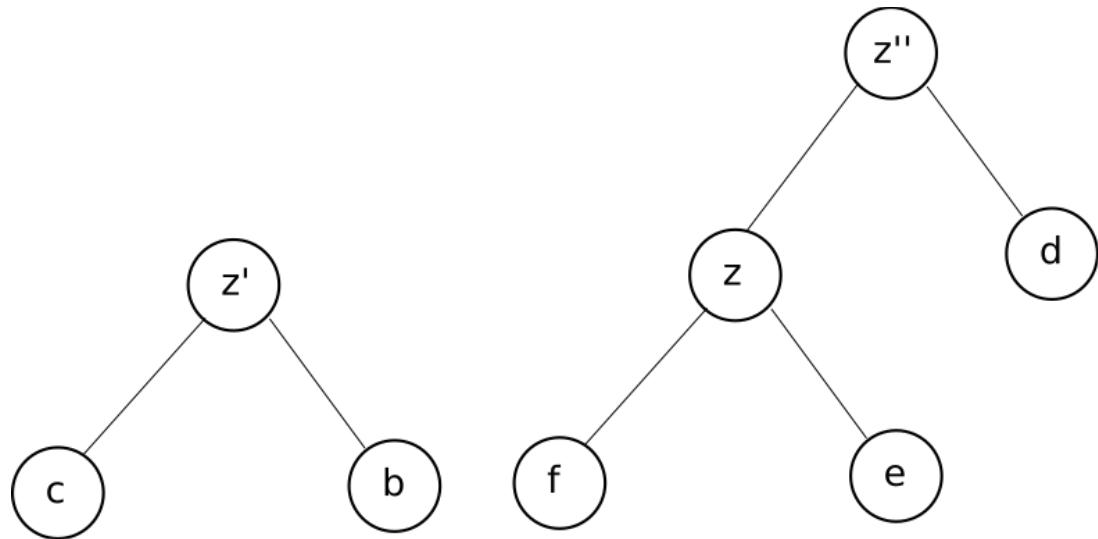
i = 3

$$Q^{(3)} = \{a:0.45, z':0.25, d:0.16, z:0.14\}$$

$$x=z$$

$$y=d$$

$$w(z'') = w(x) + w(y) = 0.14 + 0.16 = 0.25$$



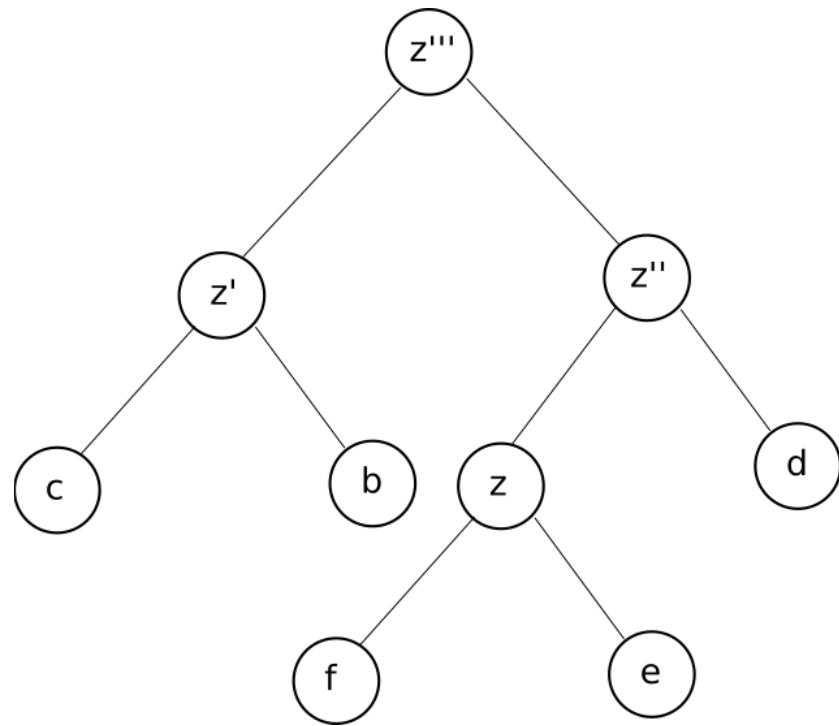
i = 4

$$Q^{(4)} = \{a:0.45, z':0.25, z'':0.3\}$$

$$x=z'$$

$$y=z''$$

$$w(z''') = w(x) + w(y) = 0.25 + 0.3 = 0.55$$



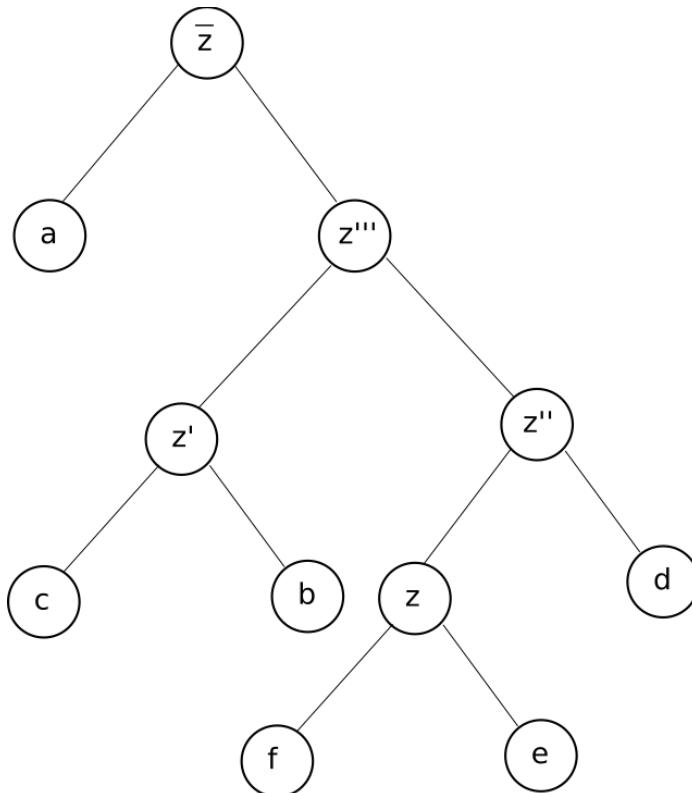
$i = 5$

$$Q^{(5)} = \{a:0.45, z''':0.55\}$$

$$x=a$$

$$y=z''''$$

$$w(\bar{z}) = w(x) + w(y) = 0.45 + 0.55 = 1.0$$

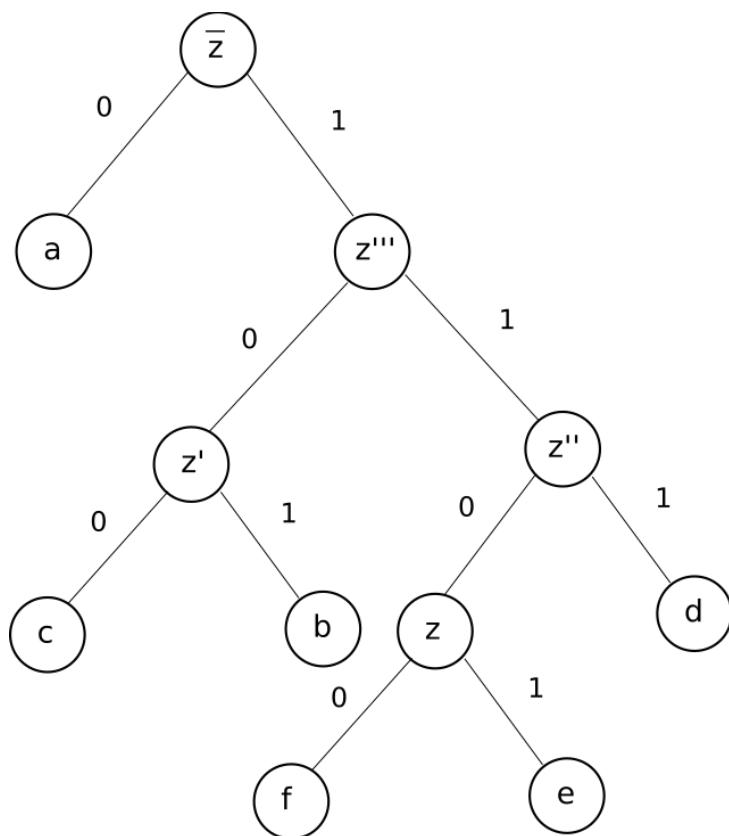


A árvore binária está completa. Como gerar os códigos dos símbolos?

Note que na árvore de Huffman, todos os símbolos são nós folhas da árvore. A ideia consiste em percorrer a árvore iniciando pela raiz. Cada vez que visitarmos um nó esquerda, adicionamos um 0 ao código e cada vez que visitarmos um nó a direita, adicionamos um 1 ao código

```
// A raiz da árvore deve ser dada
PrintCodes(node, code = '') {
    if node.left != NIL {
        code = str(code) + '0'
        PrintCodes(node.left, code)
    }
    if node.right != NIL {
        code = str(code) + '1'
        PrintCodes(node.right, code)
    }
    // Se é nó folha (símbolo)
    if node.left == NIL and node.right == NIL
        print('node: ', code)
}
```

Esse processo é equivalente a rotular as arestas da árvore de Huffman como segue:



o que nos leva aos seguintes códigos:

- a: 0
- b: 101
- c: 100
- d: 111
- e: 1101
- f: 1100

A seguir demonstraremos um teorema que garante a otimalidade do algoritmo de Huffman.

Teorema: Seja

$$l(C) = \sum_{i=1}^n w_i l(\alpha_i)$$

o tamanho esperado do código C. Então, pode-se mostrar que não existe nenhum código C cujo valor de $l(C)$ é inferior ao valor de $l(CH)$. Em outras palavras, o código de Huffman é ótimo em termos de minimizar o tamanho do código.

Prova:

De acordo com Shannon, a quantidade de informação do símbolo α_i é:

$$h(\alpha_i) = \log_2 \frac{1}{w_i} \quad (\text{logaritmo do inverso da probabilidade})$$

A entropia associada a distribuição de probabilidades dos símbolos é dada por:

$$H(w) = \sum_{i=1}^n w_i h(\alpha_i) = \sum_{i=1}^n w_i \log \frac{1}{w_i} = \sum_{i=1}^n w_i \log w_i^{-1} = -\sum_{i=1}^n w_i \log w_i$$

O valor de $H(w)$ define um limite inferior para o tamanho médio do código. Nenhum código C possui $l(C) \geq H(w)$: esse é o resultado do Teorema da Codificação de Shannon.

Iniciamos calculando a diferença entre $H(w)$ e $l(C)$.

Então, desejamos calcular a diferenças

$$H(w) - l(C) = \sum_{j=1}^n w_j \log_2 \frac{1}{w_j} - \sum_{j=1}^n w_j l(\alpha_j)$$

Note que podemos reescrever $l(\alpha_j)$ como:

$$l(\alpha_j) = \log_2 2^{l(\alpha_j)}$$

de modo que

$$H(w) - l(C) = \sum_{j=1}^n w_j \log_2 \frac{1}{w_j} - \sum_{j=1}^n w_j \log_2 2^{l(\alpha_j)} = \sum_{j=1}^n w_j \log_2 \frac{1}{w_j} + \sum_{j=1}^n w_j \log_2 2^{-l(\alpha_j)}$$

Colocando em evidência, temos:

$$H(w) - l(C) = \sum_{j=1}^n w_j \log_2 \frac{2^{-l(\alpha_j)}}{w_j}$$

Sabemos que:

$$\log_b a = \frac{\log_x a}{\log_x b}$$

ou seja,

$$\log_2 \frac{2^{-l(\alpha_j)}}{w_j} = \frac{\ln \frac{2^{-l(\alpha_j)}}{w_j}}{\ln 2} = \frac{1}{\ln 2} \ln \frac{2^{-l(\alpha_j)}}{w_j}$$

o que nos leva a:

$$H(w) - l(C) = \frac{1}{\ln 2} \sum_{j=1}^n w_j \ln \frac{2^{-l(\alpha_j)}}{w_j}$$

Mas, sabemos que $\ln x \leq x - 1$, ou seja:

$$H(w) - l(C) \leq \frac{1}{\ln 2} \sum_{j=1}^n w_j \left(\frac{2^{-l(\alpha_j)}}{w_j} - 1 \right)$$

Aplicando a distributiva, temos:

$$H(w) - l(C) \leq \frac{1}{\ln 2} \left(\sum_{j=1}^n 2^{-l(\alpha_j)} - \sum_{j=1}^n w_j \right)$$

Mas sabemos que a soma das probabilidades em uma distribuição é igual a 1:

$$H(w) - l(C) \leq \frac{1}{\ln 2} \left(\sum_{j=1}^n 2^{-l(\alpha_j)} - 1 \right) \quad (*)$$

Veremos a seguir que se C for o código de Huffman, a sequência $l(\alpha_1) + l(\alpha_2) + \dots + l(\alpha_n)$ deve satisfazer:

$$\left(\frac{1}{2} \right)^{l(\alpha_1)} + \left(\frac{1}{2} \right)^{l(\alpha_2)} + \dots + \left(\frac{1}{2} \right)^{l(\alpha_n)} = \sum_{j=1}^n 2^{-l(\alpha_j)} \leq 1$$

pois os tamanhos dos códigos $l(\alpha_j)$ denotam a profundidade dos símbolos α_i em uma árvore binária. Seja $l_{max} = \max\{l(\alpha_1), l(\alpha_2), \dots, l(\alpha_n)\}$ o maior tamanho de código na árvore gerada pelo algoritmo de Huffman. Então, como temos uma árvore binária, a soma do número de nós dos níveis intermediários é sempre menor que o número de nós da última camada. Supondo uma árvore de n níveis, temos que:

$$\sum_{j=1}^n 2^{n-j} = 2^{n-1} + 2^{n-2} + \dots + 2^0 \leq 2^n$$

Por exemplo, se $n = 5$, as 4 primeiras camadas podem ter no máximo quantos nós?

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31$$

o que é menor que o número máximo de nós na camada mais profunda da árvore, $2^5=32$. Por uma simples substituição de variáveis, podemos escrever:

$$\sum_{j=1}^n 2^{l_{\max}-l(\alpha_j)} \leq 2^{l_{\max}}$$

Dividindo tudo por $2^{l_{\max}}$, chega-se em:

$$\sum_{j=1}^n 2^{-l(\alpha_j)} \leq 1$$

Logo, voltando a equação (*), temos finalmente que:

$$H(w) - l(C) \leq 0$$

ou seja, $l(C) \geq H(w)$, o tamanho do código está limitado pela entropia da distribuição de probabilidades dos símbolos.

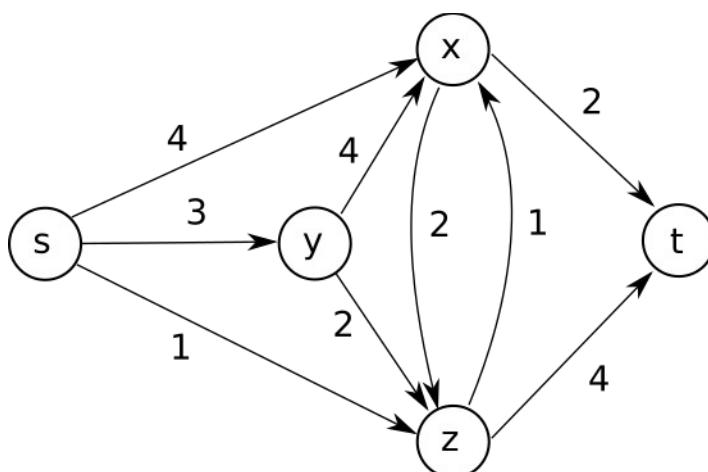
Portanto, como

$$l(C) = \sum_{i=1}^n w_i l(\alpha_i) \quad \text{e} \quad H(w) = \sum_{i=1}^n w_i h(\alpha_i)$$

a intuição consiste em notar que $l(C) = H(w)$ quando $l(\alpha_i) = h(\alpha_i)$, ou seja, quando o tamanho do código é proporcional a quantidade de informação de α_i , que é justamente a ideia do código de Huffman, pois quanto menor a probabilidade, maior a quantidade de informação e o código daquele símbolo. Lembre-se que no código de Huffman, quanto maior a quantidade de informação $h(\alpha_i)$, mais longe o símbolo α_i está da raiz, e portanto maior o seu tamanho do código $l(\alpha_i)$.

Fluxo em redes e o algoritmo de Ford-Fulkerson

Motivação: maximizar o fluxo que pode ser produzido pela fonte e consumido pelo terminal



Consideraremos grafos direcionados $G = (V, E, c)$ onde $c:E \rightarrow N^+$ é a capacidade de uma aresta.

Capacidade é a quantidade máxima de informação que pode ser transmitida por uma aresta.
Iremos classificar os vértices de G como:

- $s \in V$: source (gerador de fluxo: não entra nada, apenas sai)
- $t \in V$: terminal ou sink (absorve fluxo: não sai nada, apenas entra)
- $\forall v \in V - \{s, t\}$: nós internos (intermediários)

Em nossos exemplos iremos considerar apenas um único source e um único terminal

Def: Um fluxo s-t é uma função $f: E \rightarrow N^+$ (associa um inteiro a cada aresta) que satisfaz:

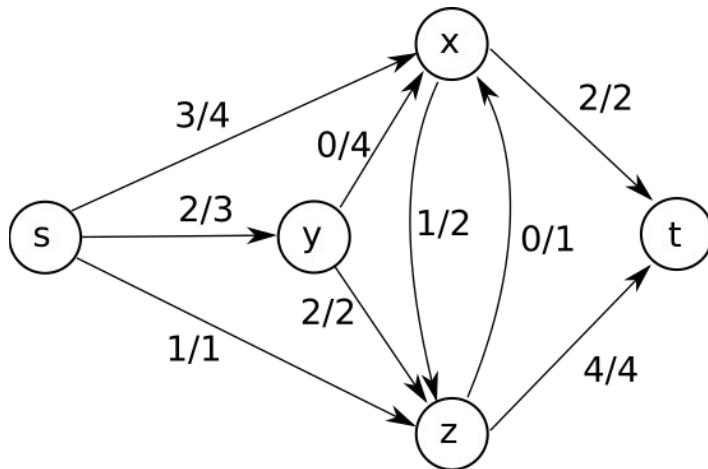
- i) $\forall e \in E, f(e) \leq c(e)$ (restrição de capacidade)
- ii) $v(f) = \sum_{e \in O(s)} f(e) = \sum_{e \in I(t)} f(e)$ (fluxo gerado na fonte é igual ao fluxo consumido no terminal)
- iii) $\forall v \in V - \{s, t\}$ (nó interno)

$$\sum_{e \in I(v)} f(e) = \sum_{e \in O(v)} f(e) \quad (\text{conservação do fluxo})$$

- iv) Limite superior: para o valor de fluxo máximo que pode circular em G

$$v(f) = \sum_{e \in O(s)} c(e)$$

Como exemplo, verifique que um fluxo válido para o grafo anterior é representado pela figura a seguir.



Basta verificar as propriedades:

- i) OK, pode-se ver facilmente que $\forall e \in E, f(e) \leq c(e)$
- ii) $\sum_{e \in O(s)} f(e) = 1 + 2 + 3 = 6 = 2 + 4 = \sum_{e \in I(t)} f(e)$ (OK)
- iii) Para $\{x, y, z\}$ temos

$$\sum_{e \in I(x)} f(e) = 3 + 0 + 0 = 1 + 2 = \sum_{e \in O(x)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(y)} f(e) = 2 = 0 + 2 = \sum_{e \in O(y)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(z)} f(e) = 1 + 1 + 2 = 0 + 4 = \sum_{e \in O(z)} f(e) \quad (\text{OK})$$

Portanto, temos de fato um fluxo válido.

Notação:

$$f^{\text{in}}(v) = \sum_{e \in I(v)} f(e) \qquad f^{\text{out}}(v) = \sum_{e \in O(v)} f(e)$$

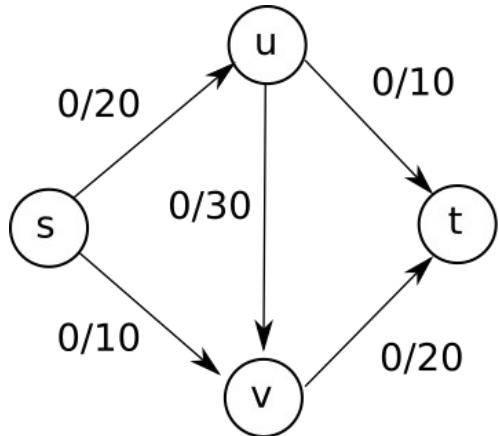
O Problema do Fluxo Máximo

Dado $G = (V, E, c)$ qual o máximo valor de $v(f)$ que pode chegar em t ?

Ideia geral

1. Condição inicial: $f(e) = 0, \forall e \in E$
2. Iteração: encontrar um caminho $s-t$ e transmitir fluxo
3. Condição de parada: Todo caminho $s-t$ encontra-se saturado

Ex:



Caso 1. $P = suvt, v(f) = 20$

Caso 2. $P_1 = sut, v(f) = 10$
 $P_2 = svt, v(f) = 10 + 10 = 20$
 $P_3 = suvt, v(f) = 20 + 10 = 30$

Note que em cada um dos casos, o valor do fluxo obtido é diferente. Não é bom que o valor do fluxo dependa da escolha dos caminhos, pois senão o algoritmo iria chegar a resultados diferentes para um mesmo problema. O ideal é que o fluxo máximo seja obtido independente da escolha dos caminhos. Para isso iremos definir o grafo residual.

Def: Grafo Residual $G_f = (V_f, E_f)$ gerado a partir de $G = (V, E, c)$

- i) $V_f = V$ (conjunto de vértices é o mesmo)
- ii) $E_f \neq E$ pois $\forall e \in E$ pode gerar até 2 arestas em E_f
 - a) Forward-Edge (FE): na mesma direção da aresta e
 $\forall e \in E$ tal que $f(e) < c(e), \exists$ em E_f uma aresta e' com capacidade residual $c(e') = c(e) - f(e)$ (exatamente o que falta para saturar)

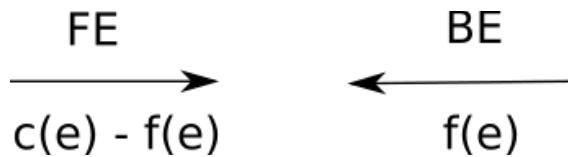
b) Backward-Edge (BE): na direção contrária a aresta e

$\forall e \in E$ tal que $f(e) > 0, \exists$ em E_f uma aresta e'' com capacidade residual $c(e'') = f(e)$ (exatamente o que já passa)

OBS: Note que no início não existem Backward-Edges pois os fluxos iniciais são nulos

RESUMO

1. $\forall e \in E$ com $f(e) = 0$ gera apenas Forward-Edge e'
2. $\forall e \in E$ com $f(e) = c(e)$ gera apenas Backward-Edge e''
3. $\forall e \in E$ com $0 < f(e) < c(e)$ gera 2 arestas: e' (FE) e e'' (BE)

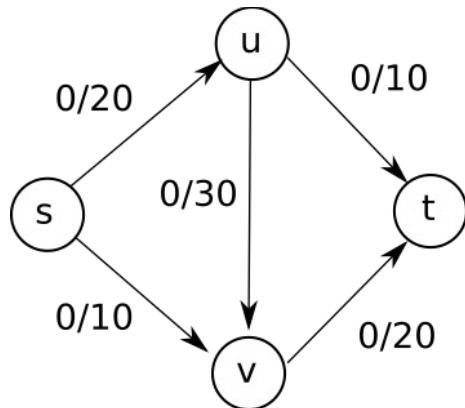


Caminhos aumentados

Para melhorar um fluxo inicial com valor f , devemos buscar por caminhos P_{st} no grafo residual $G_f = (V_f, E_f)$. A primitiva Augment que realiza essa operação de melhorar um fluxo f é dada por:

```
Augment(f, P) { (melhora fluxo f utilizando o caminho P em G_f)
    b = gargalo(P) // é o máximo que podemos passar por P
    for each e = (u, v) in P {
        if e está a favor do fluxo s-t em G (F.E.)
            f(e) = f(e) + b
        else (B.E.)
            f(e) = f(e) - b
    }
    return f
}
```

Ex:



Passo 1: Grafo residual é o próprio G

$f = 0$

$P = suvt$

$b = 20$

$$f(su) = 0 + 20 = 20$$

$$f(uv) = 0 + 20 = 20$$

$$f(vt) = 0 + 20 = 20$$

Passo 2:

$$f = 20$$

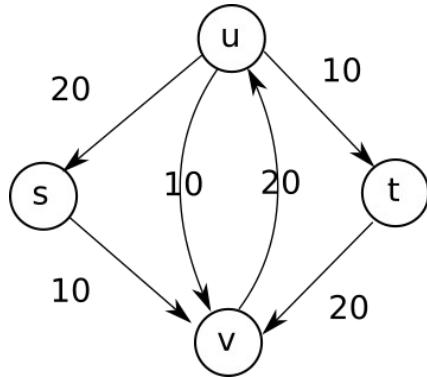
$$P = svut$$

$$b = 10$$

$$f(sv) = 0 + 10 = 10$$

Contrária a aresta (u,v)

$$f(vu) = f(uv) - 10 = 20 - 10 = 10$$

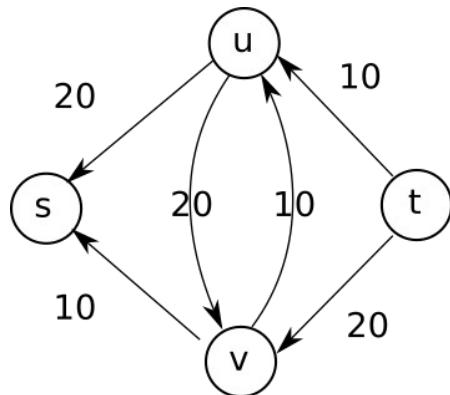


Passo 3:

$$f = 30$$

$\nexists P_{st}$ em $G_f \rightarrow$ PARE

O fluxo máximo vale 30



Teorema: O resultado da operação $f' = \text{Augment}(f, P)$ é um fluxo válido.

i) f' não excede a capacidade (no caso de F.E.)

$$f'(e) = f(e) + b \leq f(e) + (c(e) - f(e))$$

pois para qualquer aresta F.E. do caminho $c(e) - f(e)$ será maior ou igual que b

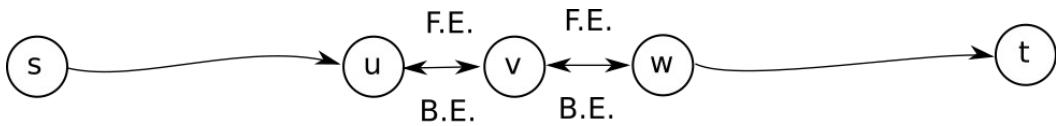
ii) f' nunca é inferior a zero (no caso de B.E.)

$$f(e) \geq f'(e) = f(e) - b \geq f(e) - f(e) = 0$$

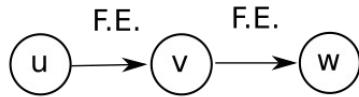
pois para qualquer aresta B.E. do caminho $f(e)$ será maior ou igual que b

iii) conservação (tudo que entra em v sai de v)

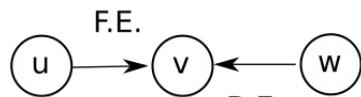
$$f^{\text{in}}(v) = f^{\text{out}}(v) \quad \text{para todo nó interno } v$$



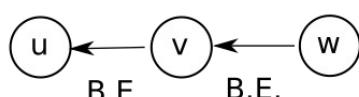
4 possibilidades



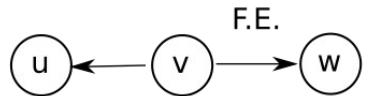
+b na entrada
+b na saída



+b na entrada
-b na saída



-b na entrada
-b na saída

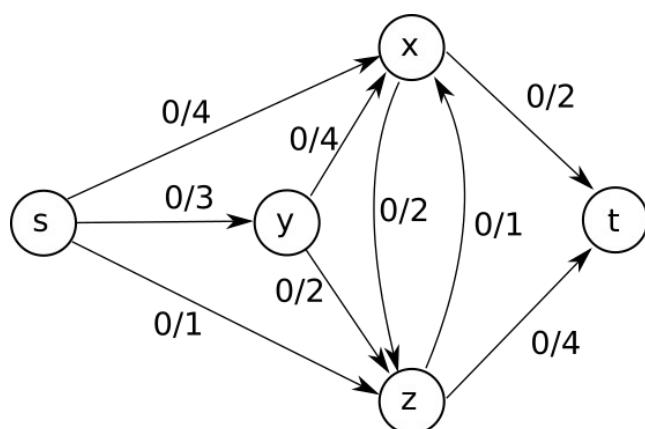


-b na saída
+b na saída

Portanto, a operação Augment(f, P) preserva fluxos e podemos usá-la para melhorar um dado fluxo. Veremos a seguir uma sequência lógica de passos para obter o fluxo máximo em um grafo: o algoritmo de Ford-Fulkerson, um algoritmo guloso pois tenta passar o máximo de fluxo possível em cada caminho aumentado.

```
Max_Flow(G, s, t, c) {
    f = 0
    for each e in E
        f(e) = 0
    while ∃Pst in Gf {
        Let Pst be one of these paths
        f' = Augment(f, Pst)
        Update the residual graph Gf
    }
}
```

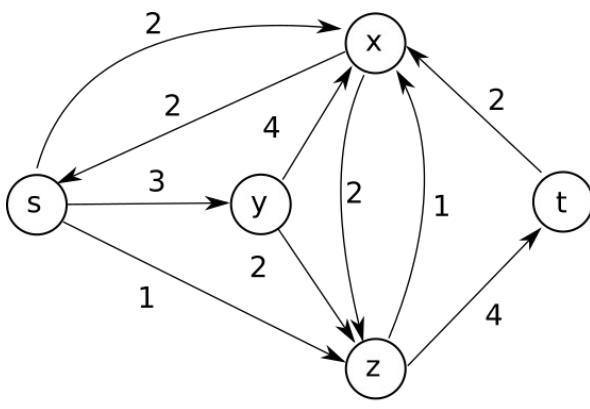
Ex: Utilizando ao algoritmo de Ford-Fulkerson, encontre o fluxo máximo



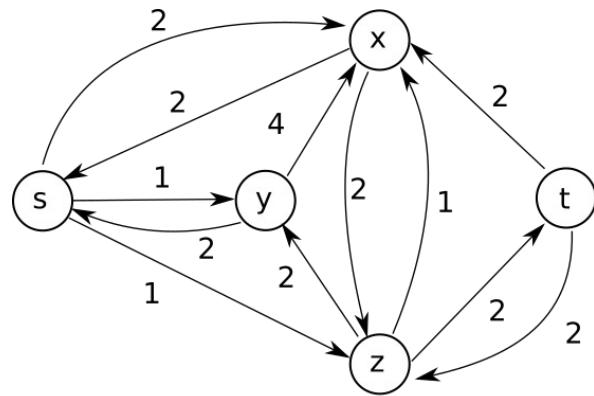
i	P_{st}	b	$f(e), \forall e \in P_{st}$	f
1	sxt	2	$f(sx) = 0 + 2 = 2$	$0 + 2 = 2$
2	syzt	2	$f(xt) = 0 + 2 = 2$	$2 + 2 = 4$
3	szt	1	$f(sy) = 0 + 2 = 2$	
4	syxzt	1	$f(yz) = 0 + 2 = 2$ $f(zt) = 0 + 2 = 2$ $f(sz) = 0 + 1 = 1$ $f(zt) = 2 + 1 = 3$ $f(sy) = 2 + 1 = 3$ $f(yx) = 0 + 1 = 1$ $f(xz) = 0 + 1 = 1$ $f(zt) = 3 + 1 = 4$	$4 + 1 = 5$ $5 + 1 = 6$

Portanto, o fluxo máximo vale 6.

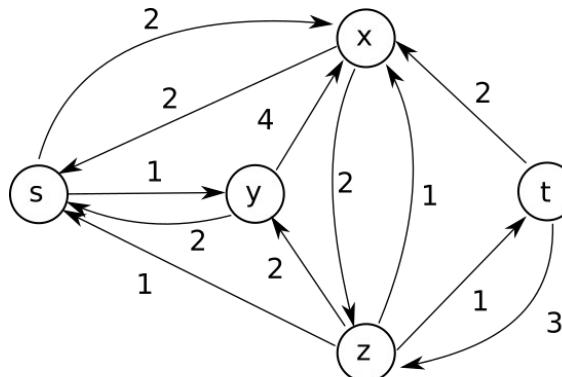
A seguir encontram-se os grafos residuais após cada um dos passos do algoritmo.



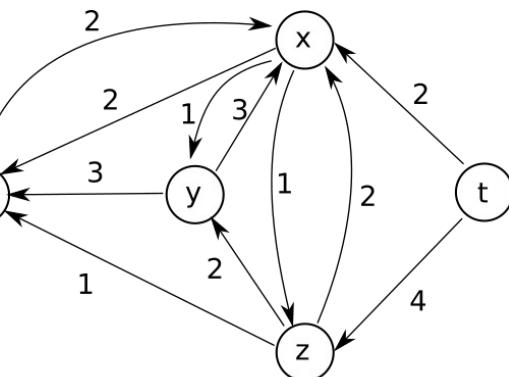
G_f Passo 1



G_f Passo 2

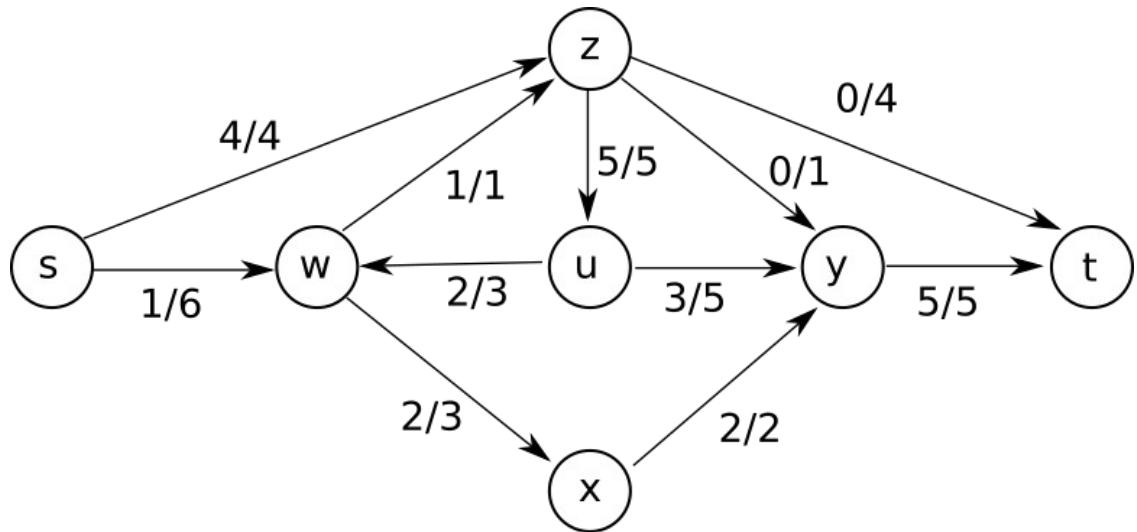


G_f Passo 3

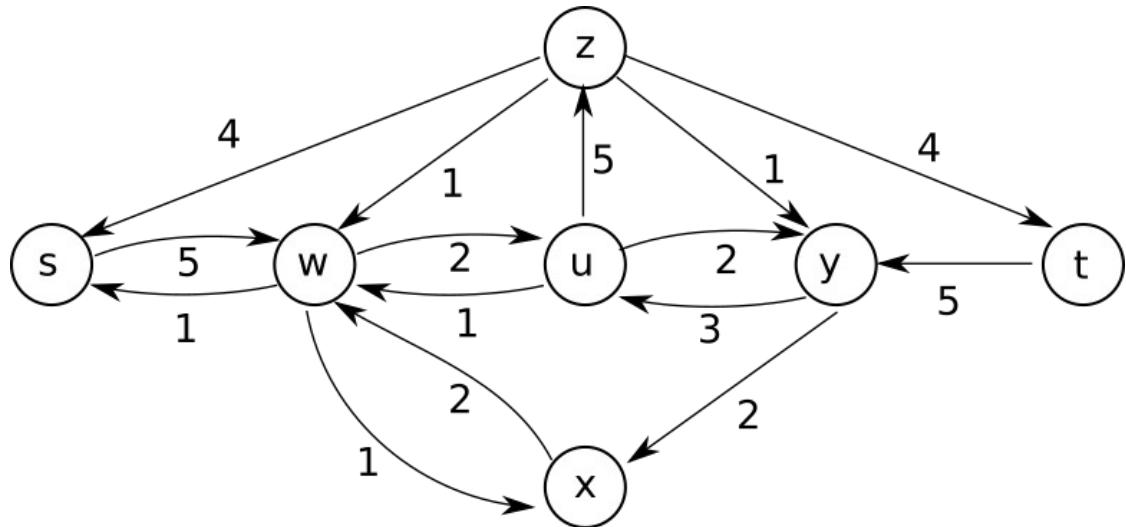


G_f Passo 4

Exercício: Dado o grafo G a seguir, responda: o fluxo dado é máximo? Justifique sua resposta.



Para verificar se o fluxo dado é máximo, devemos observar o grafo residual. Sendo assim, o grafo residual de G fica:



Note que $\exists P_{st} = swuzt$ no grafo residual, portanto o fluxo não pode ser máximo. Como o gargalo do caminho é 2, podemos aumentar o fluxo nessas arestas de duas unidades, ou seja:

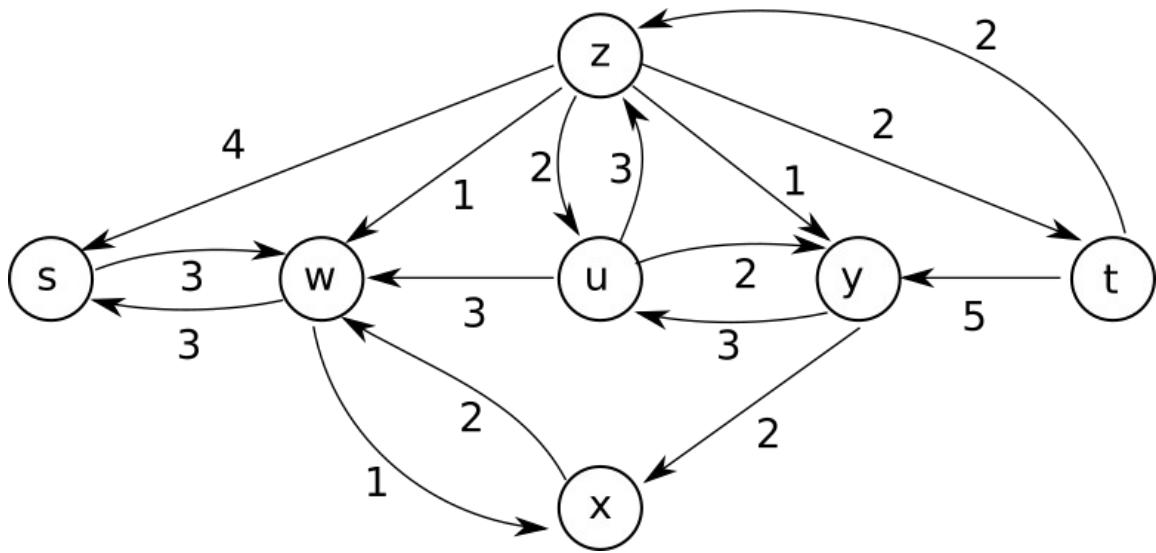
$$f(sw) = 1 + 2 = 3, \text{ pois a aresta } (s,w) \text{ está alinhada ao fluxo em } G$$

$$f(wu) = 2 - 2 = 0, \text{ pois a aresta } (w,u) \text{ está ao contrário do fluxo em } G$$

$$f(uz) = 5 - 2 = 3, \text{ pois a aresta } (u,z) \text{ está ao contrário do fluxo em } G$$

$$f(zt) = 0 + 2 = 2, \text{ pois a aresta } (z,t) \text{ está alinhada ao fluxo em } G$$

De modo que o valor total do fluxo agora é 7. O grafo residual é atualizado para:



Note que $\nexists P_{st}$ no grafo residual. Portanto, podemos concluir agora que o fluxo é máximo.

Fluxos e cortes

Aplicações: segmentação de imagens, classificação supervisionada, emparelhamentos

Motivação: cortes especificam limites superiores para $v(f)$ (valor do fluxo máximo). A relação entre fluxo e cortes nos permite resolver um problema NP-Hard (corte mínimo em grafos) em tempo polinomial.

Ideia: Ao partitionar G em 2 componentes A e B , a capacidade das arestas cortadas impõe um limite superior para o fluxo que pode sair de A e chegar em B .

Def: Um corte $s-t$ em G é qualquer partição de V em A e B tal que $s \in A$ e $t \in B$

Def: A capacidade de um corte é definida como:

$$c(A, B) = \sum_{e \in O(A)} c(e) \quad (\text{soma das capacidades das arestas que saem de } A \text{ e chegam em } B)$$

Min-cut/Max-flow: resultado que mostra a equivalência entre 2 problemas duais

Encontrar o fluxo máximo corresponde a encontrar o corte de mínima capacidade (2 problemas que podem ser resolvidos com um único algoritmo)

Para provar o teorema Min-cut/Max-flow, iremos primeiramente demonstrar uma série de resultados preliminares importantes que serão utilizados depois.

Teorema: Seja f um fluxo $s-t$ e (A, B) um corte $s-t$. Então:

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \quad (\text{ou seja, o valor do fluxo depende diretamente do corte: tudo que sai de } A \text{ menos tudo que entra em } A)$$

$$v(f) = \sum_{e \in O(s)} f(e) = f^{\text{out}}(s)$$

Sabe-se que $f^{\text{in}}(s) = 0$ então podemos escrever $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$

Mas também sabemos que $\forall v \in V - \{s, t\}$ (nó intermediário), pela conservação do fluxo vale:

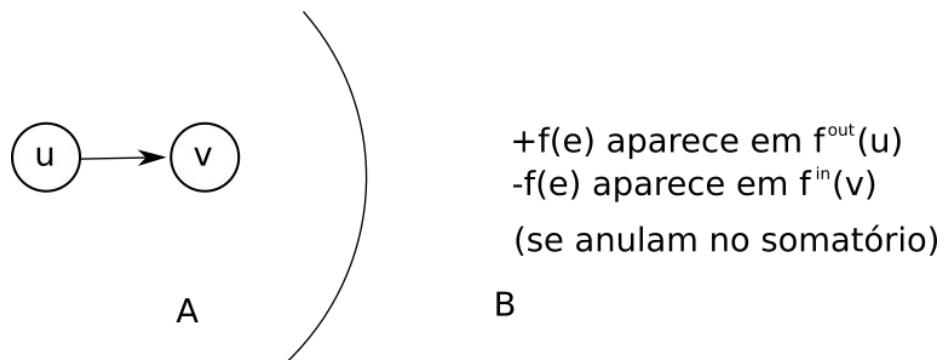
$$f^{\text{in}}(v) = f^{\text{out}}(v) \Leftrightarrow f^{\text{out}}(v) - f^{\text{in}}(v) = 0$$

Desse modo, podemos expressar $v(f)$ como:

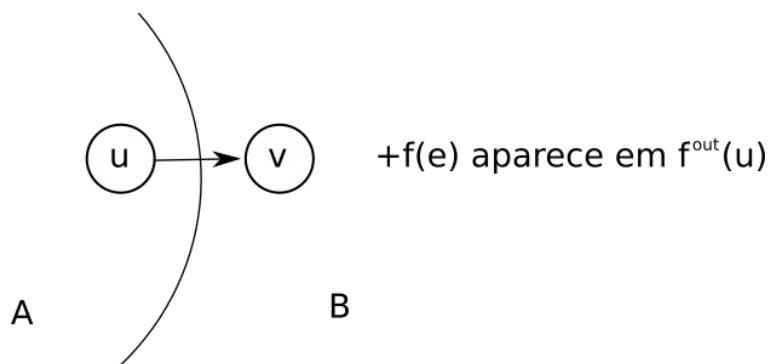
$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) \quad (\text{todos os termos desse somatório são nulos, exceto para } s)$$

Porém, existem 4 tipos de arestas em G:

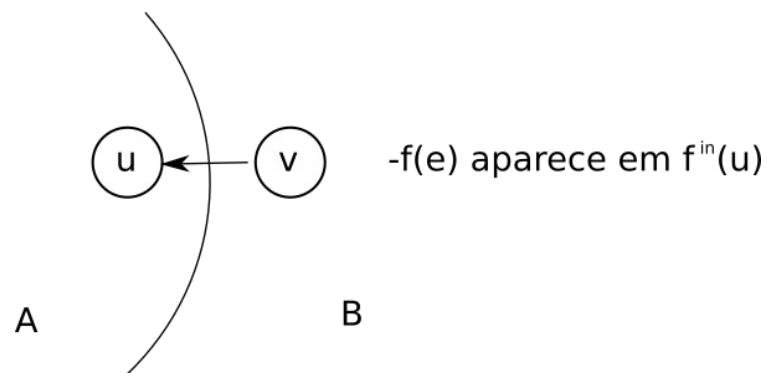
i) $e = \langle u, v \rangle$ com $u, v \in A$



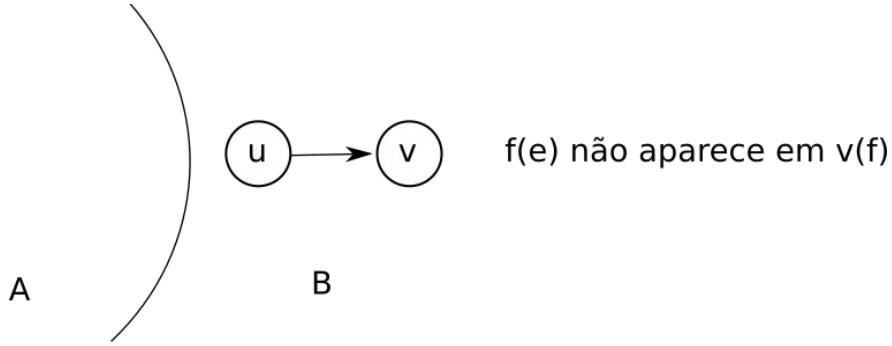
ii) $e = \langle u, v \rangle$ com $u \in A$ e $v \in B$



iii) $e = \langle v, u \rangle$ com $u \in A$ e $v \in B$



iv) $e = \langle u, v \rangle$ com $u, v \in B$



Portanto, $v(f)$ pode ser expresso por:

$$v(f) = \sum_{e \in O(A)} f(e) - \sum_{e \in I(A)} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

OBS: Note que $f^{\text{out}}(A) - f^{\text{in}}(A) = f^{\text{in}}(B) - f^{\text{out}}(B)$

Teorema: Seja um fluxo s-t qualquer e (A, B) um corte s-t. Então, $v(f) \leq c(A, B)$

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) = \sum_{e \in O(A)} f(e) \leq \sum_{e \in O(A)} c(e) = c(A, B)$$

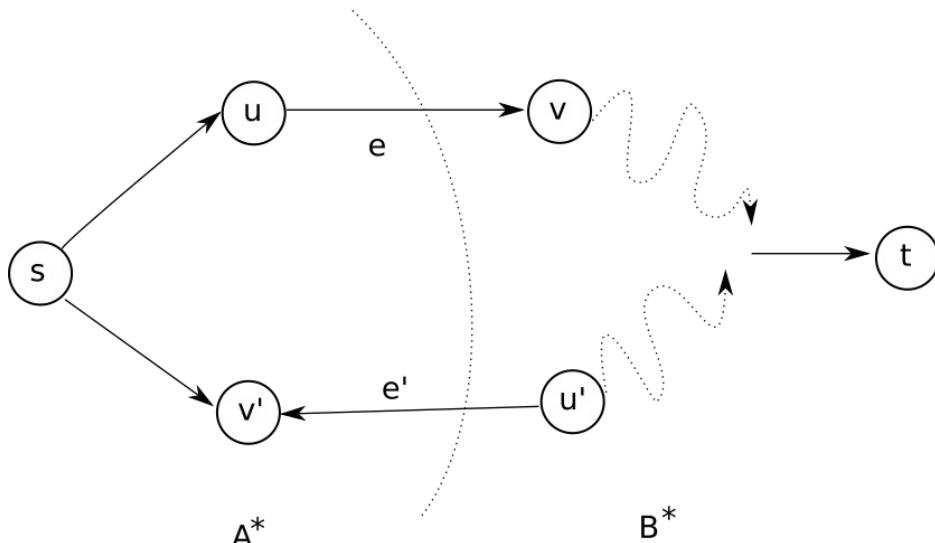
OBS: a) Se encontrarmos um fluxo s-t máximo f^* , não existe corte $c(A, B)$ com capacidade menor que $v(f^*)$. b) Se encontrarmos um corte s-t (A, B) com capacidade mínima c^* , não existe fluxo s-t com valor maior que c^*

Teorema (Min-cut/Max-flow): Seja f^* o fluxo s-t tal que $\nexists P_{st}$ em G_f (fluxo gerado pelo algoritmo Ford-Fulkerson). Então, existe um corte (A^*, B^*) para qual o valor $v(f^*) = c(A^*, B^*)$. Consequentemente, f^* tem máximo valor de fluxo e o corte (A^*, B^*) tem a mínima capacidade dentre todos os possíveis. (encontrar fluxo máximo nos dá corte mínimo, o que era NP-hard).

Prova: Seja $G_f = (V_f, E_f)$ o grafo residual de G no momento em que não existe mais caminho P_{st} . Podemos então dividir o conjunto V em 2 partições:

$$A^* = \{v \in V_f / \exists P_{sv}\} \quad (\text{atingíveis a partir de } s)$$

$$B^* = V_f - A^* \quad (\text{não atingíveis a partir de } s)$$



Note que:

a) $t \in B^*$ pois $\nexists P_{st}$ em G_f

b) Vamos analisar a aresta $e = \langle u, v \rangle$. Essa aresta certamente está saturada, ou seja, $f(e) = c(e)$, pois caso contrário haveria caminho P_{sv} uma vez que a capacidade residual da aresta F.E. correspondente em G_f seria $c(e) - f(e)$. Assim, podemos generalizar essa observação e escrever o seguinte fato:

$$\forall e \in O(A^*) \text{ tem } f(e) = c(e) \text{ (todas as arestas que saem de } A \text{ são saturadas)}$$

c) Análise da aresta $e' = \langle u', v' \rangle$. Essa aresta certamente não carrega fluxo algum ($f(e)$ nulo), pois caso contrário seria gerado uma aresta $e'' = \langle v', u' \rangle$ em G_f (B.E.) com capacidade residual $f(e')$, o que resultaria num caminho $P_{su'}$. Assim, podemos generalizar essa observação para:

$$\forall e \in I(A^*) \text{ tem } f(e') = 0 \text{ (todas as arestas que entram em } A \text{ tem fluxo nulo)}$$

Em resumo, se (A^*, B^*) é um corte, então:

$$\forall e \in O(A^*) , f(e) = c(e)$$

$$\forall e \in I(A^*) , f(e) = 0$$

Calculando o valor do fluxo temos:

$$v(f) = f^{\text{out}}(A^*) - f^{\text{in}}(A^*) = \sum_{e \in O(A^*)} f(e) - \sum_{e \in I(A^*)} f(e) = \sum_{e \in O(A^*)} c(e) = c(A^*, B^*) \quad (\text{min. capac. corte})$$

OBS: A saída do Ford-Fulkerson, $v(f)$, é de fato o valor de $c(A^*, B^*)$. Os conjuntos A^* e B^* são obtidos por uma simples busca em largura a partir de s em G_f .

“The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.”
-- Marcel Proust

Emparelhamentos em grafos bipartidos

Nesta seção, estudaremos um assunto muito importante da Teoria dos Grafos devido a sua grande aplicabilidade em diversos problemas do mundo real: os emparelhamentos em grafos bipartidos. Em resumo, sempre que precisarmos resolver problemas de alocação podemos fazer uso de algoritmos para encontrar emparelhamentos máximos em grafos bipartidos. Iniciaremos pelo estudo dos emparelhamentos estáveis e do algoritmo de Gale-Shapley. Em seguida, apresentaremos o algoritmo Húngaro e a relação entre emparelhamentos máximos e coberturas mínimas. Por fim, discutiremos como o algoritmo Húngaro pode ser empregado no problema da alocação ótima.

Emparelhamentos estáveis e o algoritmo de Gale-Shapley

Área de interface entre Teoria dos Grafos e Teoria dos Jogos.

Objetivo: Dado um grafo bipartido completo, e um conjunto de listas de preferências/rankings, encontrar dentre todos os emparelhamentos possíveis, o mais estável (noção relacionada com a ideia de evitar futuros rompimentos)

Aplicações reais em diversas áreas:

- Sistemas de recomendação (candidatos/vagas)
- Alocação de alunos a universidades
- Alocação de residentes a hospitais
- Sistema para doação de órgãos

Prêmio Nobel em 2012 na área de ciências econômicas (Lloyd Shapley) por contribuições e profundo impacto no estudo, caracterização e regulamentação de sistemas econômicos (mercados) não controlados pelo capital. (a ideia básica é que nas situações em que o algoritmo Gale-Shapley se aplica não é possível barganhar para obtenção de privilégios e vantagens impostas por escolher primeiro que os outros).

O problema do emparelhamento estável

Entrada:

$G = (V, E)$ bipartido completo $|X| = |Y| = n + 2n$ listas de preferência/rankings

Saída:

Emparelhamento perfeito S (quando $|X| = |Y|$, sempre é perfeito)

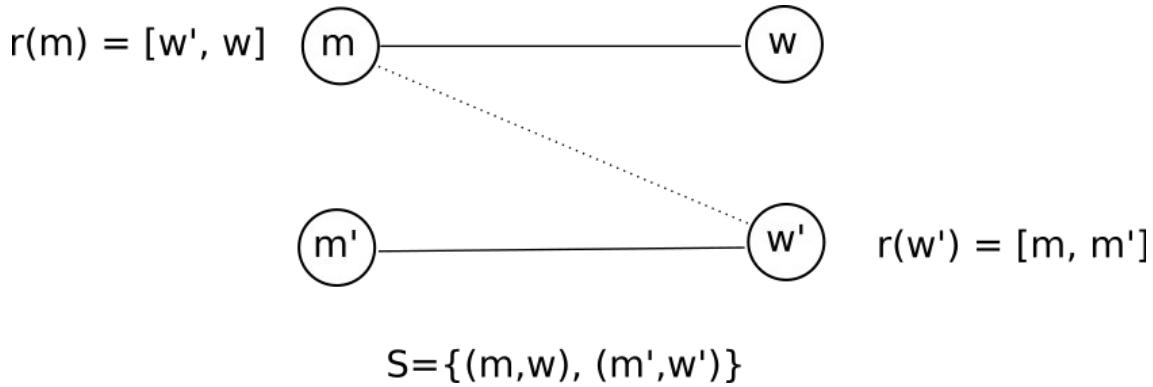
O conceito de estabilidade

- Rankings / Listas de preferências: cada objeto de um lado deve ter uma lista em que rankeia todos os elementos do outro lado

$$\begin{aligned} \forall m \in M \quad & \exists r(m) = [w_1, w_2, \dots, w_n] \\ \forall w \in W \quad & \exists r(w) = [m_1, m_2, \dots, m_n] \end{aligned}$$

Definiremos um predicado ternário $P(m, w, w')$ para denotar que m prefere w a w' , ou seja, w vem antes de w' na lista $r(m)$

Def (Estabilidade): Seja S o emparelhamento a seguir:



S é estável?

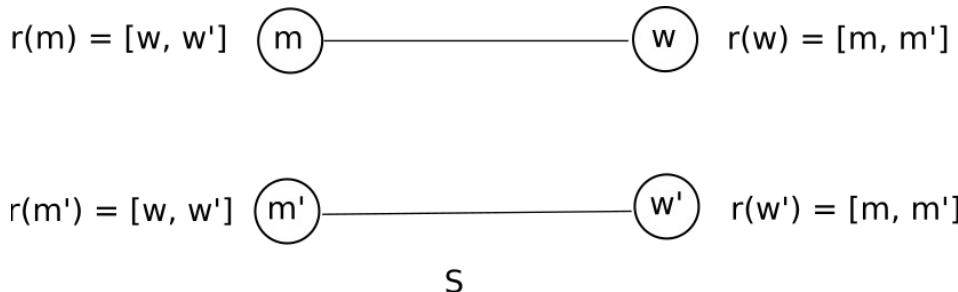
$\exists (m, w) \in S \wedge \exists (m', w') \in S$ tal que $P(m, w', w) \wedge P(w', m, m')$, portanto o par (m, w') define uma instabilidade em S pois ele não existe em S mas nada impede que venha existir. Ambos se desejam mas não estão juntos. Portanto, S não é estável.

Nada impede que m ou w' rompa o relacionamento atual de maneira unilateral na tentativa de melhorar seu ganho, ou seja, nada impede que ocorra uma ruptura e após isso o par (m, w') espontaneamente surja. Na teoria dos jogos dizemos que tanto m quanto w' podem romper unilateralmente na expectativa de maximizar o ganho e portanto a configuração atual não satisfaz o equilíbrio de Nash.

Obs: Note que $(m, w') \notin S$ (uma instabilidade nunca é um par que existe, mas sim um par que nada impede que ele possa aparecer no futuro)

Def: S é estável se S é perfeito e $\nexists (m, w)$ que provoque instabilidade

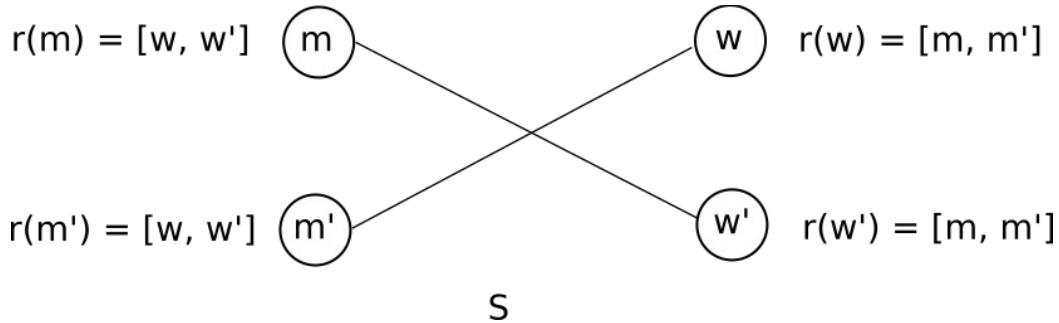
Ex: S é estável?



Sabemos que $P(m, w, w')$ e $P(w, m, m')$ e como o par (m, w) existe em S , eles não podem tentar nada melhor. Suponha que m' rompa unilateralmente com w' . Nesse caso, w vendo que m' está livre não irá largar de m para ficar com ele, pois m é o primeiro da lista. Da mesma forma, suponha que w' rompa unilateralmente com m' . O homem m não vai romper com w para ficar com w' pois prefere a parceira atual. Dessa forma, não há instabilidade que possa surgir e portanto S é estável.

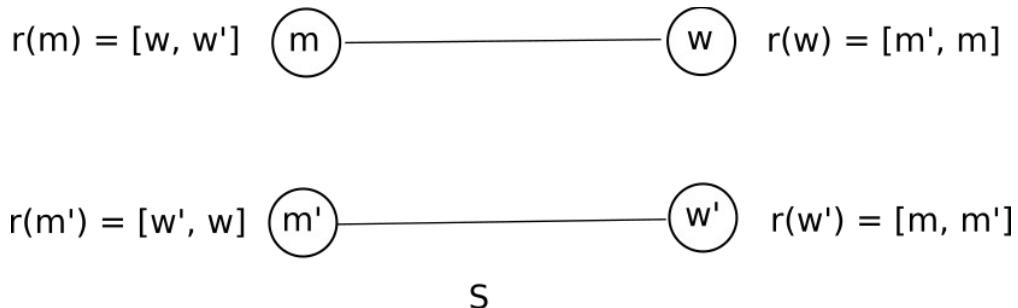
Obs: Estabilidade não é agradar a todos com primeira opção!

E se S for assim?

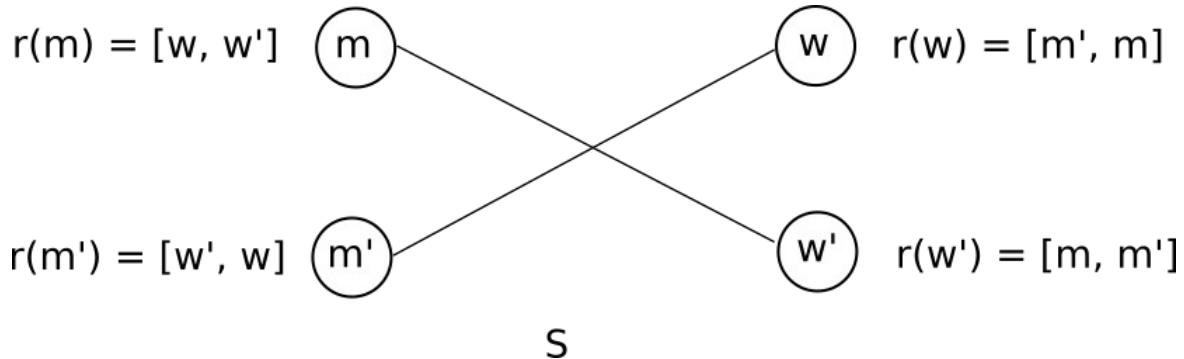


S é instável (tende a voltar para a configuração de cima). O que impede m e w de ficarem juntos no futuro? Instabilidade

Ex:



Note que para os homens está OK (perfeito), mas para mulheres está trocado. Isso significa instabilidade? Não, pois S é estável. Veja que a mulher w' deseja melhorar e pode romper unilateralmente com m' . Porém, m não se interessa por w' e a restante (m, w') nunca irá se formar. O mesmo vale para o caso de w romper unilateralmente com m . A aresta (m', w) nunca irá se formar pois m' não quer w . E o que dizer desse outro S ?



S também é estável. Porém agora satisfaz plenamente as mulheres. Da mesma forma que o caso anterior, não há como as arestas (m, w) ou (m', w') aparecerem e portanto não há instabilidade. Conceito: conjunto dominante – é o conjunto que propõe a ligação (casamento), quem toma a iniciativa

Questionamentos:

- 1) $\exists S$ estável para $\forall r(m_i)$ e $\forall r(w_i)$? Ou seja, é garantido que vai haver um S estável, independente de quais forem as listas de preferências? Pode-se mostrar que sim (teoria dos jogos)
- 2) Dados $r(m_i)$ e $r(w_i)$ $\forall i$, como construir S estável? Algoritmo de Gale-Shapley
- 3) S é único dado $r(m_i)$ e $r(w_i)$ $\forall i$ e um conjunto dominante? SIM

Essa é a grande propriedade do algoritmo de Gale-Shapley. O algoritmo GS não fornece vantagem para qual nó será o primeiro a escolher, não se pode barganhar). Independente da ordem que o conjunto dominante faz as escolhas, sempre resulta no mesmo emparelhamento. Do ponto de vista da economia, é o que proporciona a regulação de mercados.

Algoritmo de Gale-Shapley (conj. M dominante)

```

Enquanto  $\exists m_i$  livre (que ainda não tentou  $\forall w_i \in W$ )
    Seja  $m_i$  esse homem
    Seja  $w_i$  a mulher mais bem rankeada em  $r(m_i)$ 
    // (para a qual  $m_i$  ainda não propôs!)
    Se  $w_i$  está livre
         $(m_i, w_i)$  "engaged" (adiciona em S temporariamente)
    Senão
        // significa que  $\exists (\bar{m}, w_i) \in S$ 
        Se  $P(w_i, \bar{m}, m_i)$ 
             $m_i$  continua livre (vai continuar tentando)
        Senão
            // significa que  $P(w_i, m_i, \bar{m})$ 
             $(m_i, w_i)$  "engaged"
             $\bar{m}$  fica livre (vai tentar outras parceiras)

```

Propriedades do algoritmo:

- i) Algoritmo guloso: cada homem sempre propõe para sua primeira opção
- ii) Ponto de vista de $m_i \in M$: a cada troca, sequencia de parceiras piora
Para quem está no conjunto dominante, sempre que houver uma troca será para pior
- iii) Ponto de vista de $w_i \in W$: a cada troca, sequencia de parceiros melhora
Para quem está no conjunto passivo, sempre que houver uma troca será para melhorar

Análise da complexidade

Note que como o número de elementos no conjunto de homens é igual ao número de elementos no conjunto das mulheres, podemos afirmar:

- a) Há n homens livres no início e todos devem terminar com um par.
- b) No pior caso, cada homem m deve propor para n mulheres distintas.

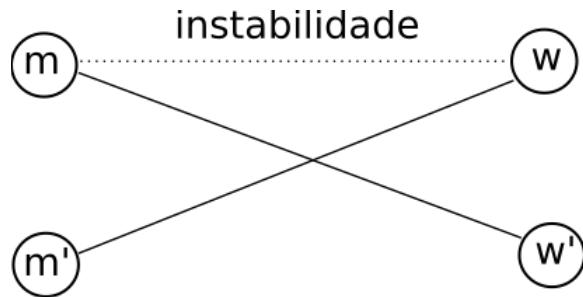
Portanto, a complexidade do algoritmo de Gale-Shapley é quadrática, ou seja, $O(n^2)$.

O resultado a seguir garante que o algoritmo de Gale-Shapley retorna um emparelhamento estável.

Teorema: O emparelhamento S retornado pelo algoritmo de Gale-Shapley é estável.

Prova: por contradição

1. Suponha que S retornado pelo algoritmo GS não é estável. Então, existe uma instabilidade (m, w)



2. Isso implica que $P(m, w, w') \wedge P(w, m, m')$

3. Mas se m finalizou o algoritmo com w' , significa que a última proposta de m foi para w'

4. Então, m propôs a w antes de w'

Isso implica que na lista de w , m' deve vir antes de m , ou seja, $P(w, m', m)$

Caso contrário, w teria ficado com m'

5. Assim, chegamos em $P(w, m, m') \wedge P(w, m', m)$, que é falso sempre pois se um dos termos é verdade o outro não pode ser verdade também. Em outras palavras, não é possível para uma mulher m preferir m a m' e m' a m simultaneamente.

6. Portanto, a suposição inicial não é válida e a instabilidade não pode existir em S .

Dessa forma, não existe S instável gerado pelo algoritmo GS.

Def: valid partner (vp)

Dizemos que w é $vp(m)$ se existe S estável tal que $(m, w) \in S$

Def: best valid partner (bvp)

Dizemos que w é $bvp(m)$ se w é $vp(m)$ e $\nexists w' \neq w$ que seja $vp(m)$ tal que $P(m, w', w)$

Teorema: Toda execução do algoritmo Gale-Shapley (com M dominante) resulta no conjunto $S^* = \{(m, bvp(m)), \forall m \in M\}$

Esse resultado é de fundamental importância pois garante que não importa a ordem de escolha dos homens, o resultado final será sempre o mesmo. Esse fato do ponto de vista da teoria dos jogos significa que nesses tipos de sistemas não existe barganha, ou seja, não é possível barganhar e comprar o direito de escolher primeiro, pois o resultado será sempre o mesmo.

Prova: por contradição

1. Suponha que S é um emparelhamento estável gerado pelo algoritmo GS em que m está emparelhado com alguém que não é sua $bvp(m) = w$

2. Então, m deve ter sido rejeitado por $w = bvp(m)$, uma vez que ele começa a propor em ordem decrescente de preferência

3. Seja m o primeiro homem para o qual isso ocorre (ser rejeitado). Ele pode ter sido rejeitado por:

a) m propôs a w , mas foi negado pois ela preferiu ficar com o atual m'

b) w rompeu com m por uma proposta melhor de um homem m'

Em qualquer caso, é certo que w está emparelhada a m' , o qual ela prefere, ou seja:

$$P(w, m', m)$$

4. Pela definição de bvp(m) existe um emparelhamento estável S' que contém o par (m, w) .

5. A pergunta é: Com quem m' estaria emparelhado em S' ?

Com uma mulher $w' \neq w = bvp(m)$

6. Note que m' não pode ter sido rejeitado por ninguém quando se engajou com w , pois m foi o primeiro a ser rejeitado (de acordo com o passo 3). Como m' propõe em ordem decrescente de preferência, então m' prefere w a w' (senão em S ele teria escolhido w' e não w). Logo:

$$P(m', w, w')$$

7. Assim, como temos $P(w, m', m) \wedge P(m', w, w')$ e o par $(m', w) \notin S'$, temos que (m', w) define uma instabilidade em S' (contradição pois S' era suposto estável)

Portanto, S deve obrigatoriamente associar todo homem m a sua $bvp(m)$.

Ex: Suponha que num site de relacionamentos existam as seguintes listas de preferências entre um grupo de rapazes e garotas.

Man	1	2	3	4	5
A1	B2	A2	D2	E2	C2
B1	D2	B2	A2	C2	E2
C1	B2	E2	C2	D2	A2
D1	A2	D2	C2	B2	E2
E1	B2	D2	A2	E2	C2

Woman	1	2	3	4	5
A2	E1	A1	B1	D1	C1
B2	C1	B1	D1	A1	E1
C2	B1	C1	D1	E1	A1
D2	A1	E1	D1	C1	B1
E2	D1	B1	E1	C1	A1

Utilizando o algoritmo de Gale-Shapley encontre:

a) um emparelhamento estável M sendo os rapazes o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.

b) um emparelhamento estável M' sendo as garotas o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.

i	m	w	engaged
1	A1	B2	yes
2	B1	D2	yes
3	C1	B2	yes
4	A1	A2	yes
5	D1	A2	no
6	D1	D2	yes
7	B1	B2	no
8	B1	A2	no
9	B1	C2	yes
10	E1	B2	no
11	E1	D2	yes
12	D1	C2	no
13	D1	B2	no
14	D1	E2	yes

$$S^{(0)} = \emptyset$$

$$S^{(1)} = \{A1, B2\}$$

$$S^{(2)} = \{A1, B2\}, (B1, D2)\}$$

$$S^{(3)} = \{C1, B2\}, (B1, D2)\}$$

$$S^{(4)} = \{C1, B2\}, (B1, D2), (A1, A2)\}$$

$$S^{(5)} = \{C1, B2\}, (B1, D2), (A1, A2)\}$$

$$S^{(6)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$$

$$S^{(7)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$$

$$S^{(8)} = \{C1, B2\}, (D1, D2), (A1, A2)\}$$

$$S^{(9)} = \{C1, B2\}, (D1, D2), (A1, A2), (B1, C2)\}$$

$$S^{(10)} = \{C1, B2\}, (D1, D2), (A1, A2), (B1, C2)\}$$

$$S^{(11)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$$

$$S^{(12)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$$

$$S^{(13)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2)\}$$

$$S^{(14)} = \{C1, B2\}, (E1, D2), (A1, A2), (B1, C2), (D1, E2)\}$$

b) Solução

i	w	m	engaged
1	A2	E1	yes
2	B2	C1	yes
3	C2	B1	yes
4	D2	A1	yes
5	E2	D1	yes

$$S^{(0)} = \emptyset$$

$$S^{(1)} = \{A2, E1\}$$

$$S^{(2)} = \{(A2, E1), (B2, C1)\}$$

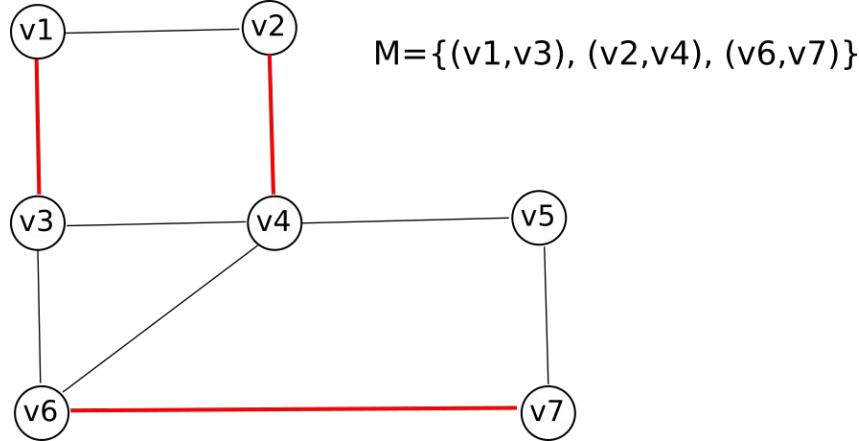
$$S^{(3)} = \{(A2, E1), (B2, C1), (C2, B1)\}$$

$$S^{(4)} = \{(A2, E1), (B2, C1), (C2, B1), (D2, A1)\}$$

$$S^{(5)} = \{(A2, E1), (B2, C1), (C2, B1), (D2, A1), (E2, D1)\}$$

Emparelhamentos e o algoritmo Húngaro

Def: Seja $G = (V, E)$ um grafo. Um emparelhamento $M \subseteq E$ é um subconjunto de arestas não adjacentes (não compartilham vértices). Em outras palavras, um emparelhamento é um conjunto independente de arestas



Def: Vértice M-saturado

É todo $v \in V$ que é extremidade de alguma aresta de M

Ex: $v1, v2, v3, v4, v6, v7$

Def: Vértice M-não-saturado

É todo vértice $v \in V$ que não é extremidade de aresta de M

Ex: $v5$

Def: Emparelhamento perfeito

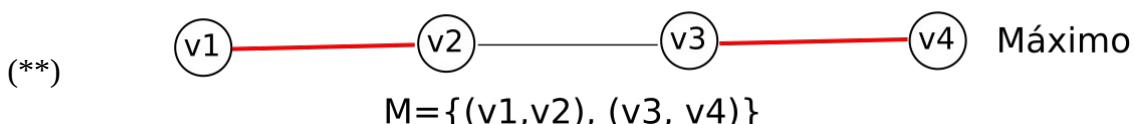
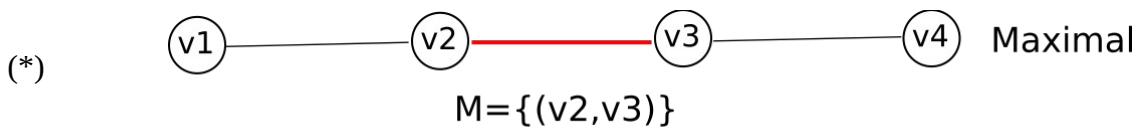
$\exists M \subset E$ tal que $\forall v \in V$ v é M-saturado $\rightarrow M$ é perfeito
(possível apenas se G tem um número par de vértices)

Def: Emparelhamento maximal

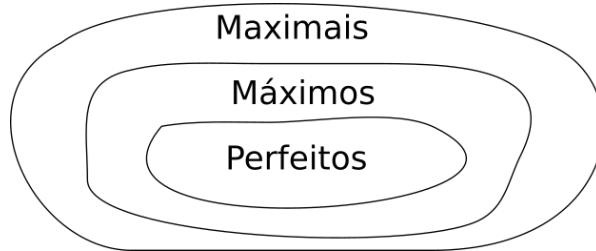
Se M não pode ser aumentado com o acréscimo de uma aresta então M é maximal

Def: Emparelhamento máximo

Se M é um emparelhamento de tamanho máximo dentre todos os possíveis, M é máximo



A figura a seguir ilustra a relação entre emparelhamentos perfeitos, máximos e maximais.



Def: Caminho M-alternado (**)

Todo caminho P em que arestas estão alternadamente em M e E – M é um caminho M-alternado

Def: Caminho M-aumentado (*)

Todo caminho M-alternado em que tanto a origem quanto o destino são M-não -saturados

Def: Diferença simétrica

Dados dois conjuntos A e B, a diferença simétrica entre eles é a operação dada por:

$$A \oplus B = (A - B) \cup (B - A)$$

Essa operação é equivalente a operação lógica “ou exclusivo” (XOR), pois retorna os elementos que estão em A ou em B mas não na intersecção. A dedução a seguir mostra essa equivalência.

$$A \oplus B = (A - B) \cup (B - A)$$

Lembre-se que $A - B \equiv A \cap \bar{B}$, o que nos leva a:

$$A \oplus B = (A \cap \bar{B}) \cup (\bar{A} \cap B)$$

Aplicando a propriedade distributiva, temos:

$$A \oplus B = (A \cup B) \cap (A \cap \bar{B}) \cup (\bar{A} \cup B) \cap (\bar{A} \cap B)$$

Sabemos que $A \cup \bar{A} = U$ e $A \cap U = A$, onde U é o conjunto universo. Assim, podemos escrever:

$$A \oplus B = (A \cup B) \cap (\bar{A} \cup \bar{B})$$

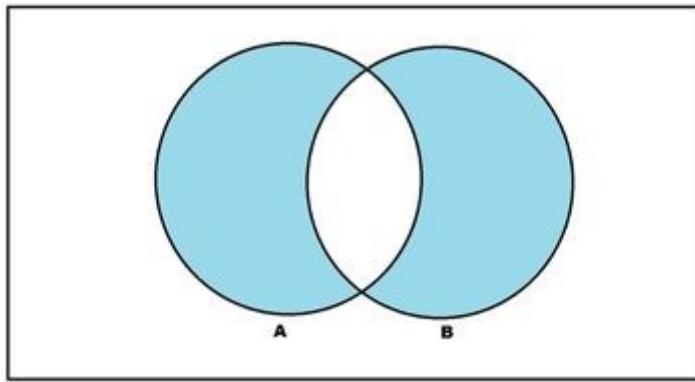
Pela lei de De Morgan, $(\bar{A} \cup \bar{B}) \equiv \overline{(A \cap B)}$, o que nos leva a:

$$A \oplus B = (A \cup B) \cap \overline{(A \cap B)}$$

Mas, $A \cap \bar{B} \equiv A - B$, o que finalmente nos leva a:

$$A \oplus B = (A \cup B) - (A \cap B)$$

mostrando que a diferença simétrica nada mais é que a união menos a intersecção, conforme ilustra a figura a seguir.



Teorema: Sejam M_1 e M_2 dois emparelhamentos distintos em G . Seja H o subgrafo definido pela diferença simétrica entre M_1 e M_2 :

$$M_1 \oplus M_2 = (M_1 - M_2) \cup (M_2 - M_1) \quad (\text{arestas que estão ou em } M_1 \text{ ou em } M_2 \text{ mas não em ambos})$$

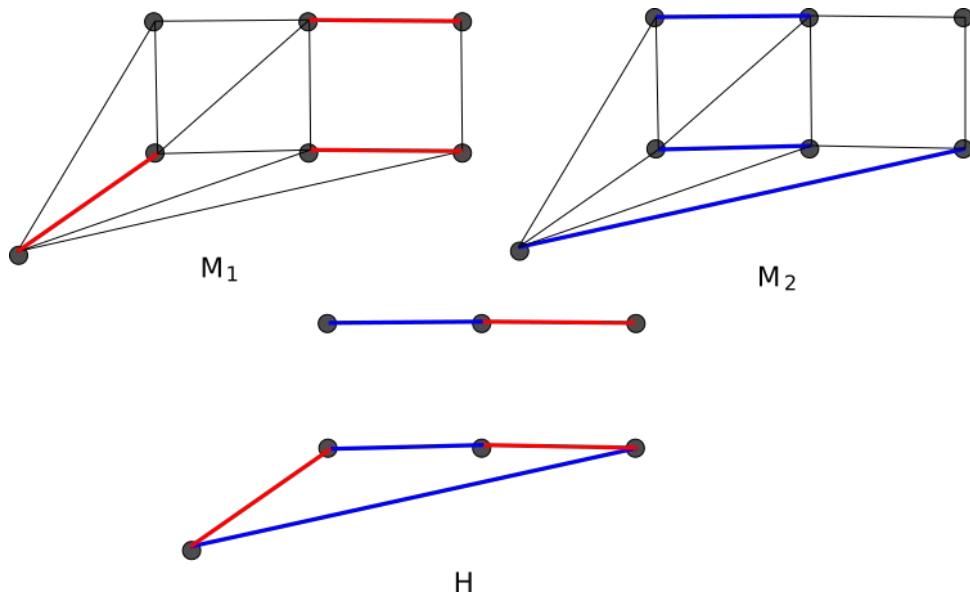
Então, cada componente conexa de H é de um de dois tipos:

- a) um ciclo de comprimento par cujas arestas estão alternadamente em M_1 e M_2 ;
- b) um caminho cujas arestas estão alternadamente em M_1 e M_2 e cujos vértices extremidades são insaturados em um dos dois emparelhamentos;

Prova: Seja v um vértice arbitrário de H . Então, temos:

- i) v é um vértice extremidade de uma aresta em $M_1 - M_2$ e também de uma aresta em $M_2 - M_1$;
- ii) v é um vértice extremidade de uma aresta de $M_1 - M_2$ ou de $M_2 - M_1$, mas não de ambos;

Em ambos os casos i) e ii), como M_1 é um emparelhamento, existe no máximo uma aresta em M_1 com o vértice v como sua extremidade. De modo similar, existe no máximo uma aresta em M_2 com o vértice v como extremidade. Assim, no caso i) v tem grau dois em H enquanto que no caso ii) v tem grau um em H . A figura a seguir ilustra o resultado desse teorema.



Pergunta: Como podemos determinar se um emparelhamento M é máximo, ou seja, que ele o melhor possível para um dado grafo G ? O resultado a seguir nos fornece a resposta.

Teorema de Berge (1957)

Um emparelhamento M em G é máximo $\Leftrightarrow G$ não possui caminho M -aumentado

(ida) $p \rightarrow q = !q \rightarrow !p$

M é máximo \rightarrow não há caminho M -aumentado = Há caminho M -aumentado $\rightarrow M$ não é máximo

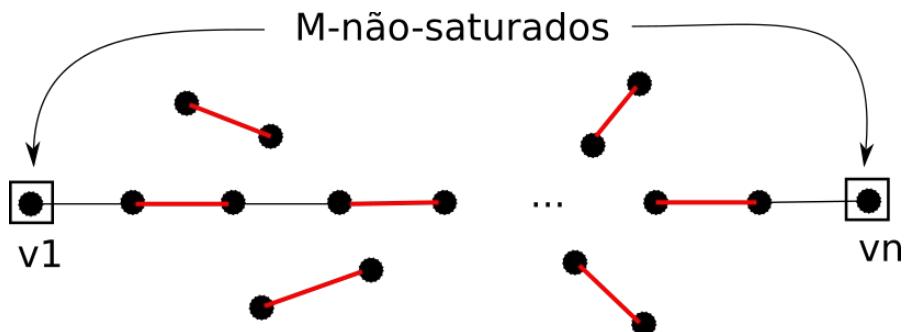
Seja um emparelhamento $M \subset E$. Então, há 2 tipos de arestas no grafo G :

- i) $\forall e \in M$: escura (faz parte de M)
- ii) $\forall e \notin M$: clara (não faz parte de M)

Suponha que P seja um caminho M -aumentado em G . Então P é da seguinte forma:

$P = \text{clara, escura, clara, escura, clara, escura, ... , escura, clara}$
 (1^a) (última)

Pela alternância das arestas, temos que para cada clara devemos ter uma escura. Então se existem m pares de arestas (clara, escura) deve haver uma última aresta clara no final, de modo que o número total de arestas de P é da forma $2m + 1$. Note que é um número ímpar não importa o valor de n (há portanto uma aresta clara a mais que escura). É importante notar que em um caminho M -aumentado o número de arestas claras é sempre maior que o número de arestas escuras.



arestas de M que não estão em P
 não podem incidir em vértices de P

Assim, podemos definir um novo emparelhamento M' como:

$$M' = M \oplus P = \{ \forall e \in M \text{ que não está em } P \} \cup \{ \forall e \in E - M \text{ que estão em } P \}$$

(escura) (clara)

de modo que $|M'| = |M| + 1$. A operação que consiste em transformar M em M' usando P é chamada de “Transferência ao longo do caminho M -aumentado P ”

A prova da volta consiste em verificar que se não há caminho M -aumentado então M deve ser um emparelhamento máximo.

(volta) $q \rightarrow p = !p \rightarrow !q$

Não existe caminho M -aumentado $\rightarrow M$ é emparelhamento máximo

Seja M' um emparelhamento máximo arbitrário em G . Deseja-se mostrar que $|M| = |M'|$, de modo que se M tiver o mesmo número de arestas que um emparelhamento máximo, ele também será máximo.

Seja H o subgrafo definido pelo conjunto de arestas dado pela diferença simétrica entre M e M' :

$$M \oplus M' = (M - M') \cup (M' - M)$$

Pelo teorema anterior, sabemos que os componentes conexos de H são:

- a) ciclos de comprimento par com arestas alternadamente em M e M' ; ou
- b) caminhos alternados com arestas de M e M' cujas extremidades são não saturadas em M ou M' ;

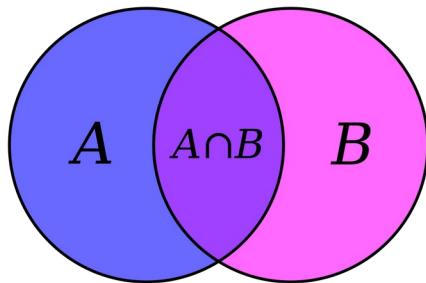
Se existe um caminho como o descrito em b) de comprimento ímpar, então tanto a origem quanto o destino são ambos não saturados em M ou ambos não saturados em M' , o que caracteriza um caminho M -aumentado (ou M' -aumentado).

Mas isso não pode ocorrer, pois de acordo com a hipótese inicial, não existe caminho M -aumentado e M' é máximo e portanto não admite caminho M' -aumentado.

Assim, os componentes de H são caminhos ou ciclos de comprimento par, de modo que contém o mesmo número de arestas de M e de M' . Matematicamente, isso implica em:

$$|M - M'| = |M' - M|$$

Da Teoria dos conjuntos, sabe-se que



$$|A| = |A - B| + |A \cap B|$$

de modo que podemos escrever:

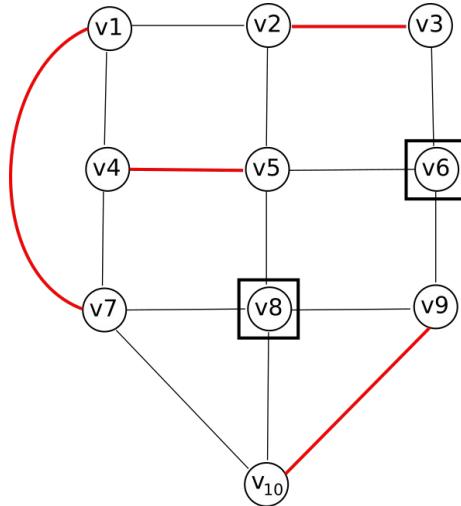
$$|M| = |M - M'| + |M \cap M'|$$

$$|M'| = |M' - M| + |M' \cap M|$$

Como $|M - M'| = |M' - M|$ segue que $|M| = |M'|$.

Portanto, o emparelhamento M é máximo, o que conclui a prova.

Ex:

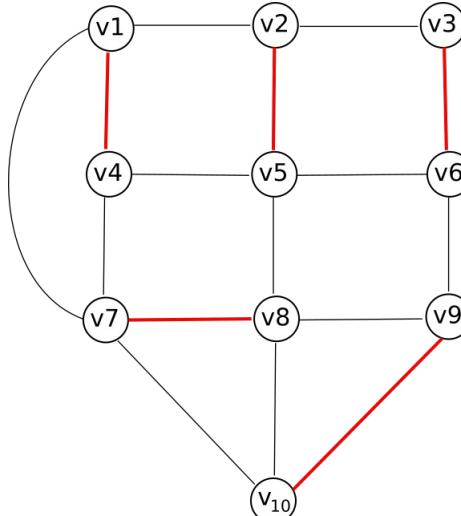


$$M = \{(v1, v7), (v4, v5)\}. M \text{ é máximo? Porque? Justifique}$$

Note que existe o caminho M-aumentado $P = v6 \rightarrow v3 \rightarrow v2 \rightarrow v5 \rightarrow v4 \rightarrow v1 \rightarrow v7 \rightarrow v8$, portanto M não é máximo. Aplicando a transferência ao longo do caminho M-aumentado P , iremos manter $(v9, v10)$ (pois não pertence ao caminho P , e inverter as arestas do caminho P , gerando:

$$M' = T(P) = T(v6 \rightarrow v3 \rightarrow v2 \rightarrow v5 \rightarrow v4 \rightarrow v1 \rightarrow v7 \rightarrow v8) = \{(v3, v6), (v2, v5), (v4, v1), (v7, v8), (v9, v10)\}$$

Note que o número de arestas em M' é uma unidade maior do que o número de arestas em M . E o novo emparelhamento M' , o que podemos dizer sobre ele?



Agora, sabemos como melhorar um emparelhamento M a partir de caminhos M-aumentados. Porém, ainda não vimos um método automatizado para buscar por caminhos M-aumentados em grafos. Esse é na verdade um problema bastante complexo para grafos arbitrários. Por motivos de simplificação, iremos adotar a hipótese de que estamos lidando com grafos bipartidos. Isso, apesar de uma limitação, não representa um problema, dado que a grande maioria dos problemas práticos envolvendo emparelhamentos são definidos em grafos bipartidos (problema do casamento, alocação, atribuição de recursos, ...). Diante do exposto, precisamos saber sob que condições um grafo bipartido admite um emparelhamento máximo, ou seja, um *matching* que sature todos os vértices da partição escolhida, por exemplo X . Para isso, iremos apresentar o teorema do casamento, que nos fornece um critério objetivo para a existência de emparelhamentos máximos em grafos bipartidos.

O Teorema do Casamento (Hall, 1935)

Seja $G = (V, E)$ um grafo bipartido com $V = X \cup Y$, $X \cap Y = \emptyset$ e $n = |X| \leq |Y|$. G contém um emparelhamento M que satura $\forall v \in X$, se e somente se, para todo subconjunto S de X tivermos:

$|N_G(S)| \geq |S|$ onde $N_G(S)$ denota o conjunto vizinhança de S no grafo G (todos elementos de G alcançáveis a partir dos elementos de S)

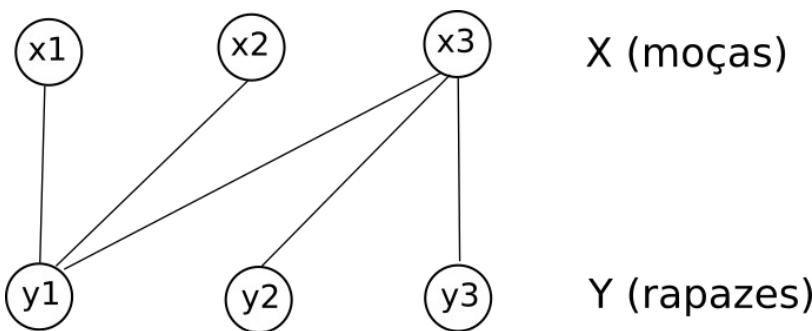
Prova:

(ida) $p \rightarrow q$ (parte fácil) (contrapositiva) $= \neg q \rightarrow \neg p$

$\exists M$ que satura todo x em $X \rightarrow \forall S \subseteq X \quad |S| \leq |N_G(S)|$

$\exists S \subseteq X \quad |S| > |N_G(S)| \rightarrow \nexists M$ que satura todo x em X

1. Vamos supor que X é o conjunto de garotas. Se existe um subconjunto de k garotas que coletivamente conhecem um conjunto de $m < k$ garotos, então é impossível casar todas elas, ou seja, não existe emparelhamento que satura todo x em X .



O número de subconjuntos de um conjunto X é o número de elementos do conjunto potência denotado por $2^{|X|}$, que nesse caso é igual a 8. Esse é um dos problemas com esse resultado: enumerar todos os possíveis subconjuntos é computacionalmente inviável para n grande (complexidade $O(2^n)$)

$$S_1 = \{x_1\}, N(S_1) = \{y_1\}$$

$$S_2 = \{x_2\}, N(S_2) = \{y_1\}$$

$$S_3 = \{x_3\}, N(S_3) = \{y_1, y_2, y_3\}$$

$$S_4 = \{x_1, x_2\}, N(S_4) = \{y_1\}$$
 (Falhou)

Como há duas garotas que conjuntamente só conhecem um único garoto, nesse grafo não será possível encontrar um emparelhamento M que sature todos os vértices de X (impossível).

(volta) $q \rightarrow p$ (prova por indução em X , mais complicada)

$\forall S \subseteq X \quad |S| \leq |N_G(S)| \rightarrow \text{Existe } M \text{ que satura todo } x \text{ em } X$

Caso base: P(1): X contém um único elemento x , ou seja, $|X| = 1$.

Nesse caso, é trivial notar que, pela hipótese inicial, x tem pelo menos um vizinho em Y , digamos y . Então, $(x, y) \in E$ uma aresta do emparelhamento M que satura todo x (x é único)

Portanto, P(1) é verdade.

Passo de indução: $P(k-1) \rightarrow P(k)$ para k arbitrário.

Supor que para X arbitrário com $|X| < k$, existe M que satura todo x em X . Queremos mostrar que isso implica que para X com $|X| = k > 1$, existe M' que satura todo x em X .

Seja $x \in X$ arbitrário. Então, pela hipótese inicial (q), existe $y \in Y$, tal que $(x, y) \in E$. Seja H o subgrafo induzido pelo conjunto de vértices $(X - x) \cup (Y - y)$, ou seja:

$$H = G[(X - x) \cup (Y - y)] \quad (H \text{ tem esses vértices e todas as arestas de } G \text{ com extremidades neles})$$

Note que $|X - x| < |X| = k$.

Caso 1: Neste caso, supomos que existe um emparelhamento M' que satura todo vértice de $X - x$. Assim, fazendo

$$M = M' + (x, y)$$

temos um emparelhamento que satura todo vértice de X . Provamos que $P(k-1) \rightarrow P(k)$.

Caso 2: Neste caso não é possível saturar todos os vértices de $X - x$ pois

$$\exists S' \in X - x \quad |S'| > |N_H(S')|$$

Defina um outro grafo induzido pelo conjunto de vértices $S' \cup N_G(S')$, chamado F_1 :

$$F_1 = G[S' \cup N_G(S')]$$

Note que S' é um subconjunto de $X - x$ e portanto também é um subconjunto de X .

Pela hipótese de indução, a cardinalidade de S' deve ser menor ou igual a cardinalidade da vizinhança de S' em G , ou seja: $|S'| \leq |N_G(S')|$.

Precisamos garantir que S' tem menos elementos que X , ou seja, $|S'| < |X| = k$. Isso é trivial, pois S' é subconjunto de $X - x$ (tem um elemento a menos), ou seja, no máximo $|S'| = k - 1$.

A última pergunta antes de aplicarmos a hipótese de indução é: será que o grafo F_1 satisfaz a condição de Hall? A resposta é SIM. Porque? S' é um subconjunto de X e o grafo G satisfaz a condição de Hall, ou seja, para todo subconjunto S' de X , $|S'| \leq |N_G(S')|$.

Podemos invocar a hipótese de indução e concluir que $\exists M'$ que satura todo x em S' no grafo F_1

Agora, note que:

1. $|S'| \leq |N_G(S')|$
2. $|S'| > |N_H(S')|$

A cardinalidade de S' passa de menor ou igual que a cardinalidade de seus vizinhos em G para maior que a cardinalidade de seus vizinhos em H . O que muda de $N_G(S')$ para $N_H(S')$ é apenas o vértice y que foi deletado de G , o que implica dizer que em G , $|S'| = |N_G(S')|$ (ao reduzir em uma unidade passou a ser estritamente maior). Se fosse menor, não poderíamos inverter a desigualdade com apenas um vértice.

Seja F_2 o grafo induzido pelo conjunto de vértices $(X - S') \cup (Y - N_G(S'))$, ou seja:

$$F_2 = G[(X - S') \cup (Y - N_G(S'))]$$

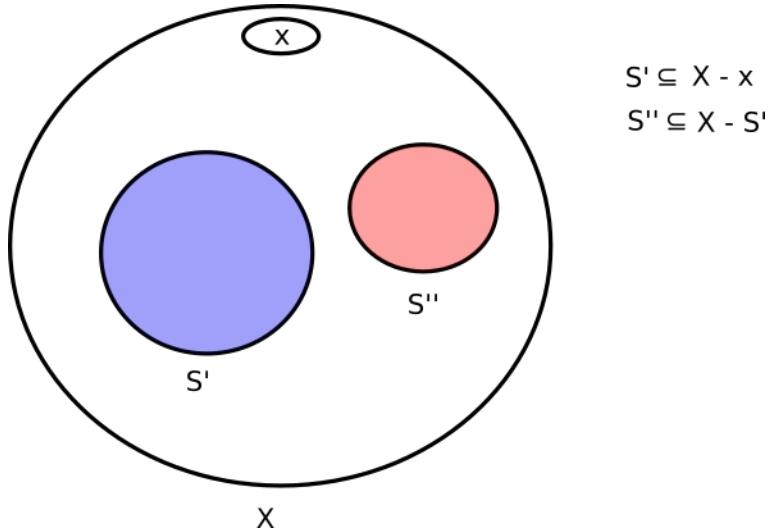
Repare que esse grafo contém tudo o que ainda resta ser emparelhado, uma vez que encontramos um emparelhamento M' que satura todo x em S' no grafo F_1 .

Note que para criar F_2 , reduzimos o número de elementos em S' e o número de elementos de $N_G(S')$ do mesmo valor, uma vez que $|S'|=|N_G(S')|$. Neste ponto, deve ser óbvio que $|X-S'|<|X|=k$ (pois S' é não vazio) e $|X-S'|=|Y-N_G(S')|$. O que nos resta agora é mostrar que o grafo F_2 satisfaz a condição de Hall. Para isso, seja $S''\subseteq X-S'$ um subconjunto arbitrário. Desejamos mostrar que a cardinalidade de S'' é menor ou igual a cardinalidade de $N_{F_2}(S'')$ (isso significa que a condição de Hall é válida).

Note que o conjunto $S'\cup S''$ é um subconjunto de X . Pela hipótese inicial, sabemos que:

$$|S'\cup S''|\leq|N_G(S'\cup S'')| \quad (\text{pois } q \text{ é verdade em } q \rightarrow p) (*)$$

Como $S'\subseteq X-x$ e $S''\subseteq X-S'$, temos que S' e S'' são subconjuntos disjuntos.



Isso implica que

$$|S'\cup S''|=|S'|+|S''|$$

Note ainda que o lado direito de (*) pode ser expresso como:

$$|N_G(S'\cup S'')|=|N_G(S')|+|N_{F_2}(S'')|$$

pois dessa forma estamos contando todos os vizinhos de S' em G , bem como todos os vizinhos de S'' que não são vizinhos de S' . Podemos garantir que não estamos contando duas vezes vizinhos em comum de S' e S'' (que estariam na intersecção) pois por construção subtraímos os vizinhos de S' do grafo F_2 , ou seja, $|N_G(S')|$ e $|N_{F_2}(S'')|$ são conjuntos disjuntos. Isso nos leva a:

$$|S'|+|S''|\leq|N_G(S')|+|N_{F_2}(S'')|$$

Mas como $|S'|=|N_G(S')|$ (verificamos isso anteriormente), temos:

$$|S''|\leq|N_{F_2}(S'')|$$

mostrando que o grafo F_2 satisfaz a condição de Hall. Aplicando a hipótese inicial (q), podemos concluir que existe um emparelhamento M'' que satura todo x em $X - S'$ no grafo F_2 . Portanto, o emparelhamento $M=M'\cup M''$ satura todo vértice x de X no grafo G . A prova está concluída.

Em palavras, se um conjunto de n garotas conhece um conjunto de n rapazes a pergunta é: quando é possível que todas se casem. A condição nos diz que é possível que todas se casem se todo subconjunto de k garotas conhece coletivamente pelo menos k rapazes.

Dado um grafo G bipartido, sabemos decidir se existe um emparelhamento máximo M ou não. Agora, iremos responder a pergunta: como buscar caminhos M -aumentados?

Árvore M-alternada

- Estrutura utilizada para buscar caminhos M-aumentados. Uma árvore T é M-alternada se satisfaz:
 - i) a raiz x_0 é M-não-saturada
 - ii) $\forall v \in T$ o único caminho de x_0 a v é M-alternado

Veremos a seguir um método que combina o teorema de Berge, o teorema do casamento e árvores M-alternadas para a obtenção de emparelhamentos máximos em grafos bipartidos em tempo polinomial: trata-se do algoritmo Húngaro.

O Método Húngaro

O padrão de crescimento de uma árvore M-alternada com raiz x_0 é tal que em qualquer estágio temos uma árvore de um de 2 possíveis tipos:

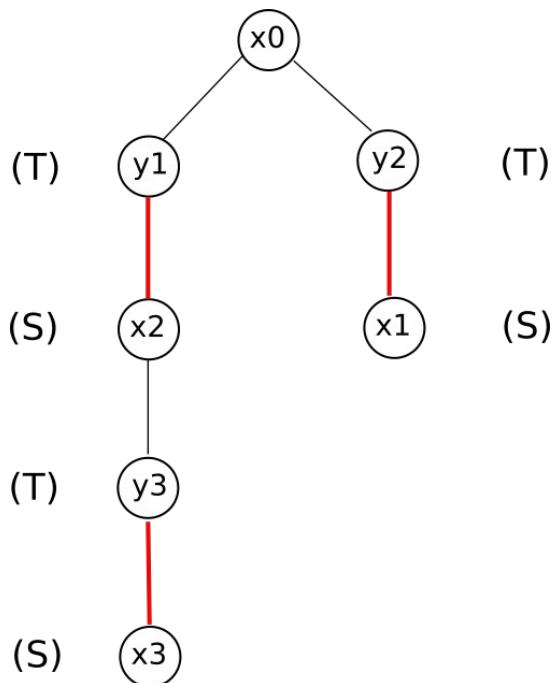
- i) Árvore do tipo-I: todos os vértices de T são M-saturados (com exceção da raiz x_0)
 - ii) Árvore do tipo-II: T possui um vértice folha M-não-saturado

Veremos a seguir como as árvores do tipo-I estão relacionadas com o teorema do casamento

Análise das árvores do tipo-I

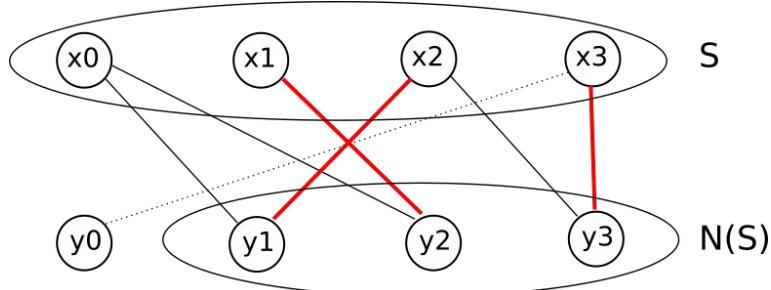
Sejam

- a) S': conjunto dos vértices a uma distância par de x_0 (raiz)
b) T: conjunto dos vértices a uma distância ímpar de x_0 (raiz)



Então, $|S'| = |T|$ (pois se todos os vértices são saturados, para cada elemento de Y tem que haver um de X correspondente – outro lado da aresta do emparelhamento). (Todo vértice a uma distância par de x_0 colocamos em S e todo vértice a distância ímpar de x_0 colocamos em T)

Define-se $S = S' \cup \{x_0\}$. Assim, $T \subseteq N(S)$ (olhando em G). Isso significa que pode ou não haver mais arestas para “pendurar” na árvore T , pois $N(S)$ engloba todos os y que estão na árvore e mais alguns que ainda podem não ter sido “pendurados”.



Desse modo podemos dividir as árvores do tipo-I em 2 subcasos:

- $T = N(S)$ (não tem mais o que adicionar a árvore, impossível crescer T)
- $T \subset N(S)$ (posso continuar crescendo a árvore pois há arestas para adicionar)

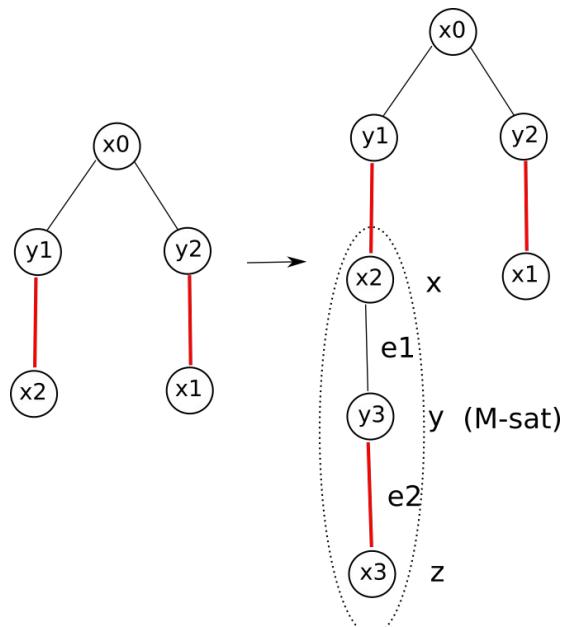
Subcaso a): condição de parada

Se $T = N(S)$, então $|N(S)| = |T| = |S'| = |S| - 1$ (menos 1 por causa da raiz). Portanto, temos que $|N(S)| < |S|$, o que fere o teorema do casamento, pois para que exista um emparelhamento M que sature $\forall v \in X$, temos que ter $|N(S)| \geq |S|$. Em outras palavras, não há como melhorar o emparelhamento M atual. Devemos parar.

Subcaso b): existe caminho M -aumentado, continuar buscando

$$T \subset N(S) \Rightarrow |N(S)| > |T| \Rightarrow |N(S)| > |S'| \Rightarrow |N(S)| > |S| - 1 \Rightarrow |N(S)| \geq |S| \quad (\text{T. C.})$$

É possível melhorar emparelhamento M buscando caminho M -aumentado (há o que adicionar na árvore T). Nesse caso, $\exists y \in G$ que não está na árvore e é adjacente a algum $x \in S$. Porém, y pode ser de 2 tipos: M -saturado ou M -não-saturado. Se y é M -saturado então \exists aresta $yz \in M$. Assim, crescemos a árvore adicionando 2 arestas: (x, y) e (y, z) , gerando uma nova árvore do tipo-I.



Caso contrário, se y é M-não-saturado, então encontramos um caminho M-aumentado P da raiz x_0 até y ($M' = T(P)$) - árvore do tipo-II. Em resumo, a ideia do algoritmo Húngaro consiste em observar repetidamente as seguintes condições:

- Tipo-I**
 - $T = N(S) \rightarrow \nexists M$ que satura $\forall v \in X$ (fere T. C.)
 - $T \subset N(S) \rightarrow$ continue buscando caminho M-aumentado P
- Tipo-II** Encontramos caminho M-aumentado P : $M' = T(P)$

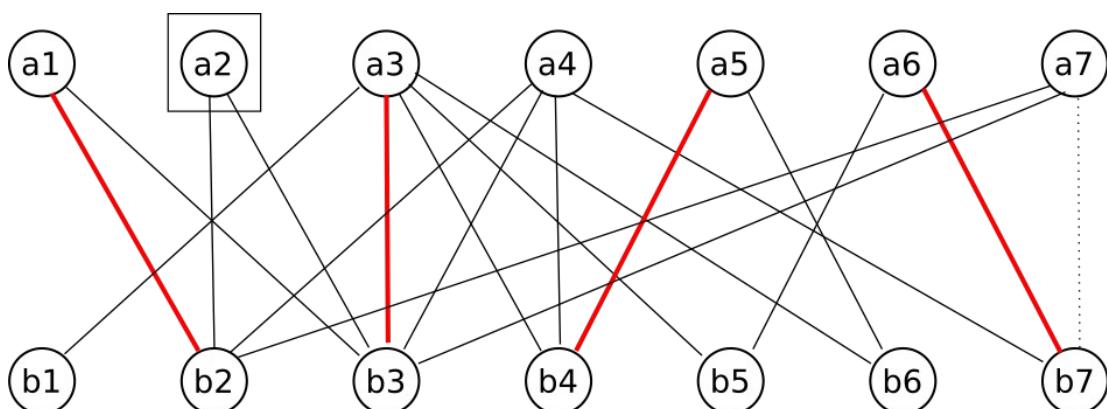
Algoritmo Húngaro

Entrada: $G = (V, E)$ bipartido + M inicial

Saída: M que satura $\forall v \in X$ ou S que fere T.C.

1. Se M satura $\forall v \in X$, pare e retorne M
Caso contrário, seja x_0 o 1º vértice M-não-saturado de X ainda não escolhido
Faça $S = \{x_0\}$ e $T = \emptyset$
2. Se $N(S) = T$, então sabemos que $|N(S)| < |S|$. Pare, pois S fere T. C. Retorne S
Caso contrário, escolha y como o 1º vértice da lista $N(S)$ tal que $y \notin T$ (não pertence a T)
3. Se y é M-saturado, seja $yz \in M$ a aresta que emparelha y a z .
Faça $S = S \cup \{z\}$, $T = T \cup \{y\}$ e volte para 2.
Se y é M-não-saturado, temos uma árvore do tipo-II e P de x_0 a y é um caminho M-aumentado. Faça $M' = T(P)$ (transf. ao longo do caminho P) e retorne para 1.

Ex: Um novo projeto a ser desenvolvido na empresa *Boogle* consiste num conjunto de 7 tarefas (a1, a2, a3, a4, a5, a6, a7). A empresa possui na equipe de desenvolvimento apenas 7 profissionais (b1, b2, b3, b4, b5, b6, b7). A partir de uma mapa de competências o gerente do projeto elaborou um modelo baseado em grafo para visualizar quais profissionais estão aptos a realizar quais tarefas. Sabendo que um profissional deve se associar a uma única tarefa, é possível alocar tarefas a pessoas de modo a completar todas as tarefas simultaneamente? Em caso positivo, forneça a alocação resultante. Em caso negativo, explique porque não é possível. (considere o matching inicial dado a seguir). Resolva o problema usando o algoritmo Húngaro.



$$M^{(0)} = \{a1b2, a3b3, a5b4, a6b7\}$$

i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
1	[a2]	\emptyset	[b2, b3]	b2 (M-sat)	(b2, a1)
	[a1, a2]	[b2]	[b2, b3]	b3 (M-sat)	(b3, a3)
	[a1, a2, a3]	[b2, b3]	[b1, b2, b3, b4, b5, b6]	b1	---

b1 (quem fez com que ele surgiu?) a3 - b3 (quem fez com que ele surgiu?) a2

$$P' = b1 \ a3 \ b3 \ a2 \rightarrow P = \text{inv}(P') = a2 \ b3 \ a3 \ b1$$

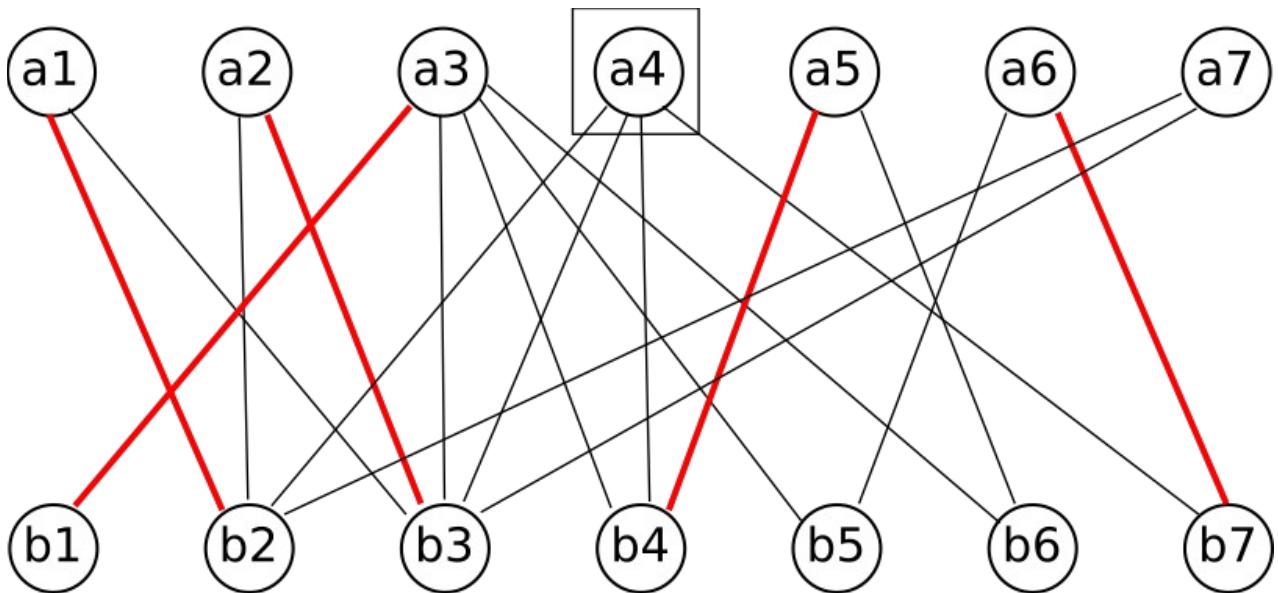
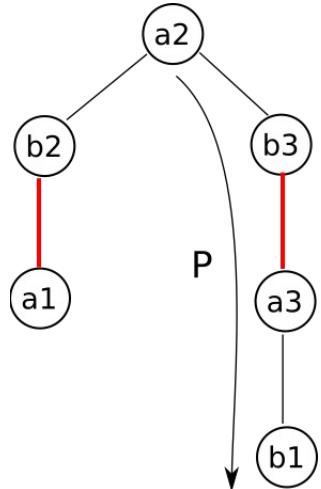
$M^{(1)} = T(P)$ (percorrer P ligando as arestas que não estão em $M^{(0)}$ e desligando as que estão)

a2 b3 = OK

b3 a3 = x

a3 b1 = OK

$$M^{(1)} = \{a1b2, a2b3, a3b1, a5b4, a6b7\}$$



i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
2	[a4]	\emptyset	[b2, b3, b4, b7]	b2 (M-sat)	(b2, a1)
	[a1, a4]	[b2]	[b2, b3, b4, b7]	b3 (M-sat)	(b3, a2)
	[a1, a2, a4]	[b2, b3]	[b2, b3, b4, b7]	b4 (M-sat)	(b4, a5)
	[a1, a2, a4, a5]	[b2, b3, b4]	[b2, b3, b4, b6, b7]	b6	---

$$P' = b6 \ a5 \ b4 \ a4 \rightarrow P = \text{inv}(P') = a4 \ b4 \ a5 \ b6$$

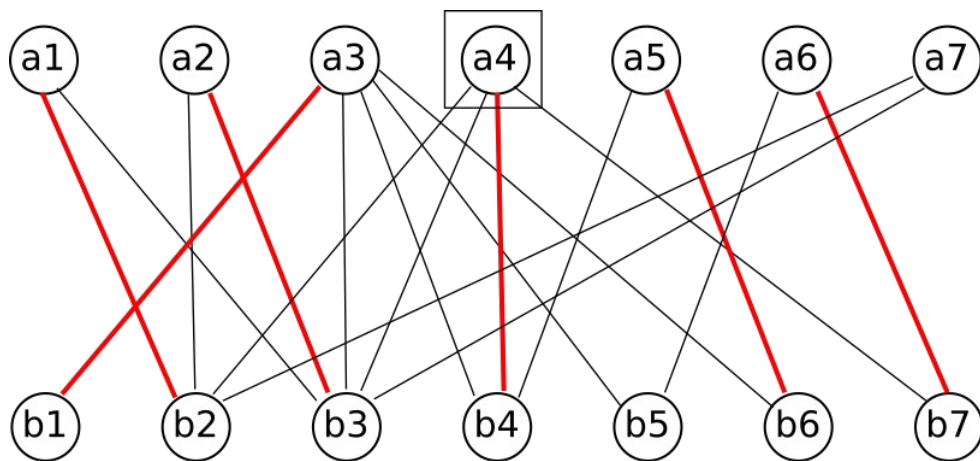
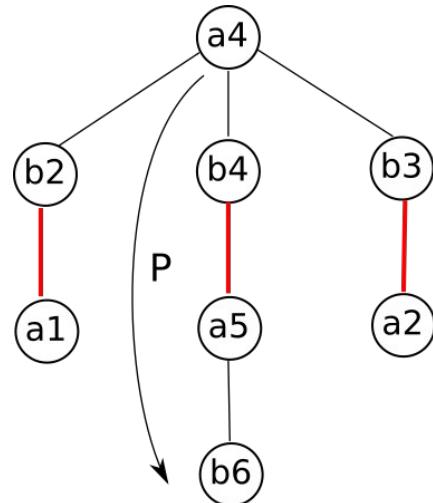
$M^{(2)} = T(P)$ (percorrer P ligando as arestas que não estão em $M^{(1)}$ e desligando as que estão)

a4 b4 = OK

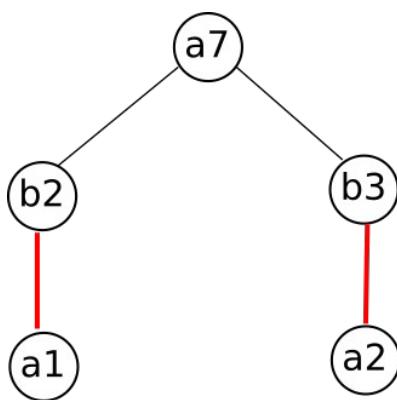
b4 a5 = x

a5 b6 = OK

$$M^{(2)} = \{a1b2, a2b3, a3b1, a4b4, a5b6, a6b7\}$$

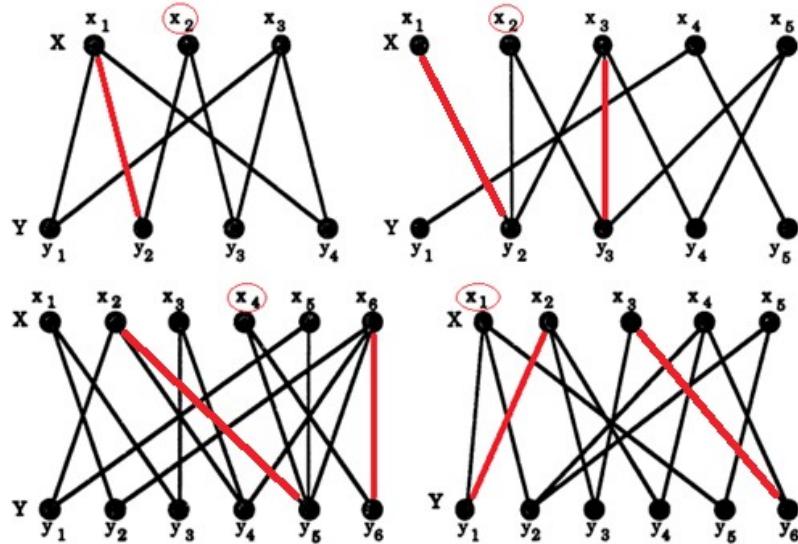


i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
3	[a7]	\emptyset	[b2, b3]	b2 (M-sat)	(b2, a1)
	[a1, a7]	[b2]	[b2, b3]	b3 (M-sat)	(b3, a2)
	[a1, a2, a7]	[b2, b3]	[b2, b3]	---	---



Como $N(S) = T$, temos que $S = \{a_1, a_2, a_3\}$ fere o T. C. pois $N(S) = \{b_2, b_3\}$. Portanto, não há emparelhamento M que sature todo vértice de X . (se houvesse aresta a_7b_7 , OK)

Ex: Utilizando o algoritmo Húngaro, encontre emparelhamentos máximos para os grafos bipartidos a seguir, ou mostre que ele não existe, indicando um subconjunto S de X que viola o Teorema do Casamento.



A seguir veremos como emparelhamentos máximos estão relacionados com coberturas mínimas.

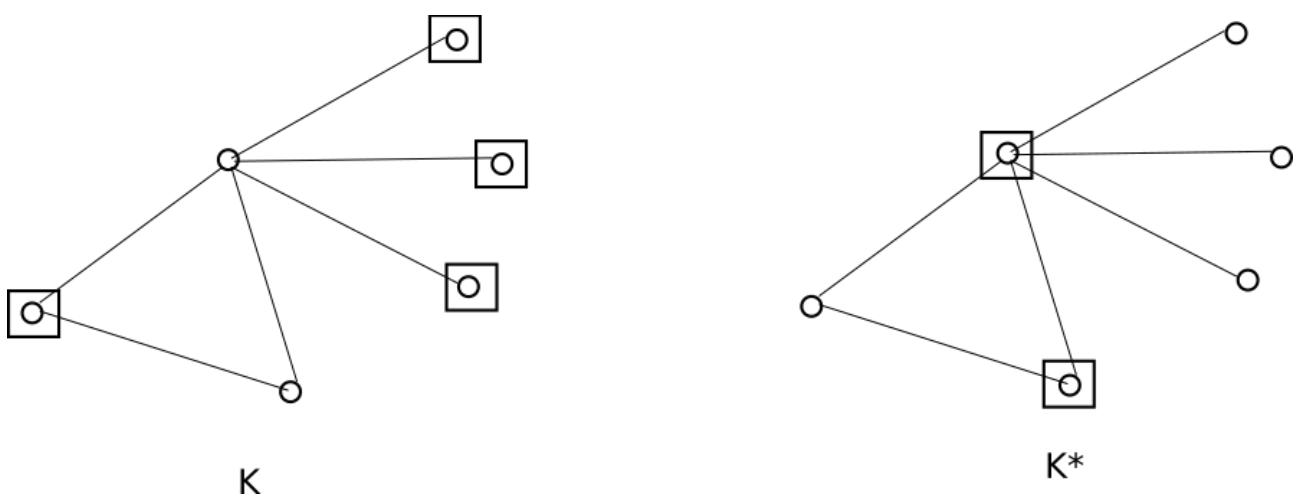
Def: O problema da cobertura (vertex cover)

Dado $G = (V, E)$ deseja-se encontrar um subconjunto $V' \subseteq V$ tal que toda aresta $e \in E$ possua uma extremidade em V' . Formalmente, uma cobertura V' de G é um subconjunto de V tal que:

$$\forall e = (u, v) \in E (u \in V' \vee v \in V')$$

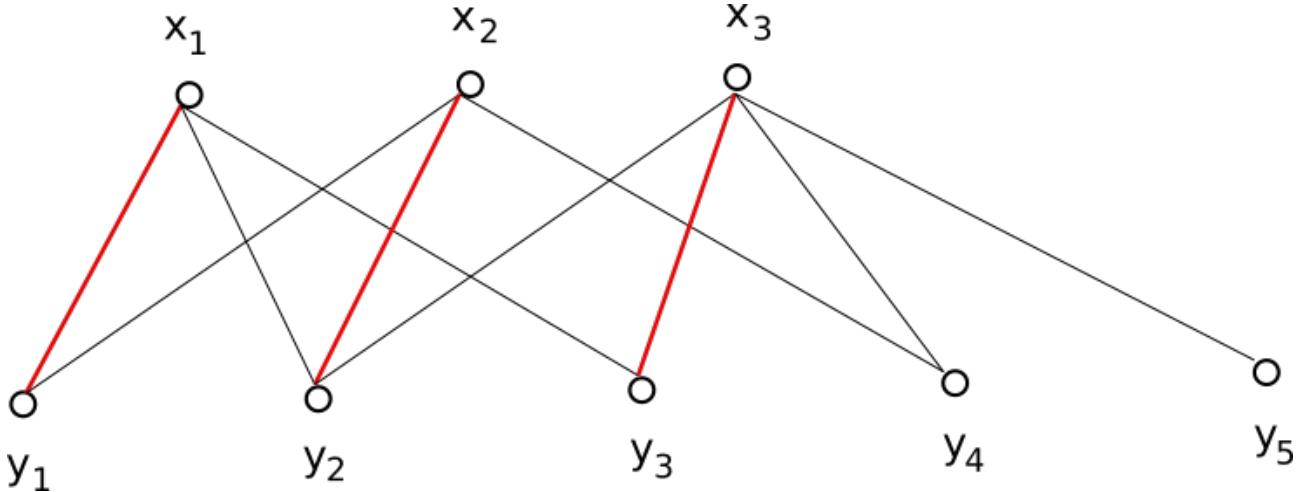
Def: Cobertura mínima

Uma cobertura de vértices K é mínima se $G = (V, E)$ não admite nenhuma outra cobertura K' com $|K'| < |K|$, ou seja, é a menor cobertura possível.



Tanto K quanto K^* são coberturas válidas para o grafo em questão, mas K^* é uma cobertura mínima pois não existe outra cobertura com menos de 2 vértices.

Existe uma relação entre emparelhamentos e coberturas. Note que o conjunto dos vértices extremidades das arestas de um emparelhamento M define uma cobertura K . A figura a seguir ilustra essa ideia.



O emparelhamento M em questão é dado por:

$$M = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$$

A cobertura K referente a esse emparelhamento é:

$$K = \{x_1, x_2, x_3, y_1, y_2, y_3\}$$

Note que o número de elementos em K é sempre maior ou igual que o número de elementos em M . Se K é uma cobertura de G e M é um emparelhamento em G , então K contém pelo menos uma extremidade de cada aresta de M . Assim, para todo K e M , temos:

$$|K| \geq |M|$$

Teorema: Seja M um emparelhamento e K uma cobertura tal que $|M| = |K|$. Então, M é um emparelhamento máximo e K é uma cobertura mínima.

Prova: Seja M^* um emparelhamento máximo e K^* uma cobertura mínima. Então:

$$|M| \leq |M^*|, \forall M \quad \text{e}$$

$$|K^*| \leq |K|, \forall K$$

Como $|M^*| \leq |K^*|$, temos:

$$|M| \leq |M^*| \leq |K^*| \leq |K|$$

Mas, se $|M| = |K|$, temos:

$$|M| = |M^*| = |K^*| = |K|$$

e portanto, como $|M^*|=|K^*|$, temos que M^* é máximo e K^* é mínimo.

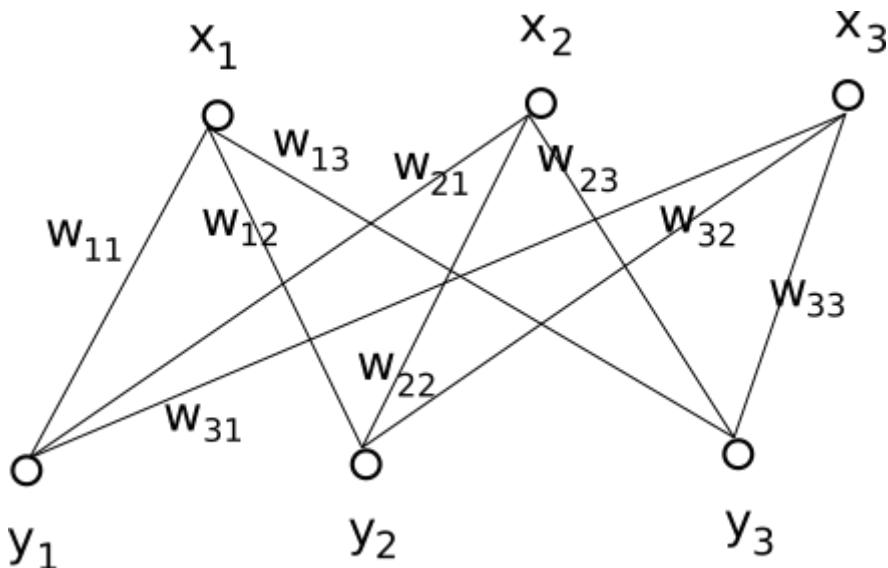
Em outras palavras, o número de arestas de um emparelhamento máximo é igual ao número de vértices de uma cobertura mínima.

O problema da alocação ótima (The optimal assignment problem)

Dado um grafo bipartido $G = (V, E)$ com $V = X \cup Y$ em que $|X| = |Y| = n$, deseja-se obter o emparelhamento máximo M que maximiza:

$$w(M) = \sum_{(v_i, v_j) \in M} w(v_i, v_j)$$

onde $w(v_i, v_j) = w_{ij}$ denota o custo da aresta (v_i, v_j) . A figura a seguir ilustra uma instância do problema da alocação ótima.



Neste exemplo, o conjunto X é composto por pessoas e o conjunto Y é composto por tarefas. Os pesos w_{ij} são os ganhos de alocar a pessoa x_i para executar a tarefa y_j (fitness). Por exemplo, se a pessoa x_i já possui experiência com a tarefa y_j , então o ganho/fitness w_{ij} é alto, pois ela já sabe ou tem noção do que deve ser feito. Porém, se a pessoa x_i precisa ser treinada para realizar a tarefa y_j , então o ganho/fitness w_{ij} é baixo, pois ela precisará de um tempo maior para realizar a tarefa em questão. Note que o conjunto de todos os pesos w_{ij} compreende uma matriz de pesos quadrada $n \times n$, onde n é o número de vértices em X (ou Y , já que devem ser iguais).

O algoritmo de Kuhn-Munkres (Húngaro)

Esse algoritmo recebe como entrada uma matriz quadrada com os pesos das arestas de um grafo bipartido completo.

Objetivo: encontrar o emparelhamento perfeito que forneça a alocação ótima.

A matriz de entrada W é da seguinte forma:

$$\begin{matrix} & x_1 & & & & x_n & \\ & \downarrow & & & & \downarrow & \\ x_1 & & & & & & \\ & \downarrow & & & & \downarrow & \\ x_2 & & w_{ij} & & & & \\ & \downarrow & & & & & \\ \dots & & & & & & \\ & \downarrow & & & & \downarrow & \\ x_n & & & & & & \\ & \downarrow & & & & \downarrow & \\ y_1 & y_2 & \dots & & & y_n & \end{matrix}$$

Durante a execução do algoritmo, manipulamos uma estrutura chamada de matriz de excessos (excess matrix). Ela possui as mesmas dimensões da matriz W, porém seus elementos são computados de forma diferente.

$$\vec{u} = \begin{matrix} v_1 & v_2 & \dots & v_n & = \vec{v} \\ \downarrow & \downarrow & & \downarrow & \\ u_1 & & & & \\ & \downarrow & & & \\ u_2 & & e_{ij} = u_{ij} + v_{ij} - w_{ij} & & \\ & \downarrow & & & \\ \dots & & & & \\ & \downarrow & & & \\ u_n & & & & \end{matrix}$$

onde $\vec{u} = (u_1, u_2, \dots, u_n)$ é o vetor de pesos dos vértices em X e $\vec{v} = (v_1, v_2, \dots, v_n)$ é o vetor de pesos dos vértices em Y. Durante a execução do algoritmo, sempre temos a seguinte desigualdade:

$$w_{ij} \leq u_i + v_j$$

o que define um limite superior para o valor de w_{ij} . Com isso, temos que a matriz de excessos é não negativa, ou seja, $\forall i, j \quad e_{ij} \geq 0$.

O algoritmo tenta iterativamente reduzir a soma total dos u_i 's e v_j 's.

Inicialização dos pesos

$$v_j = 0 \quad \text{para } j = 1, 2, \dots, n$$

$$u_i = \max\{w_{ij}, j = 1, 2, \dots, n\} \quad \text{para } i = 1, 2, \dots, n \quad (\text{maior peso na } i\text{-ésima linha})$$

Ao longo das iterações:

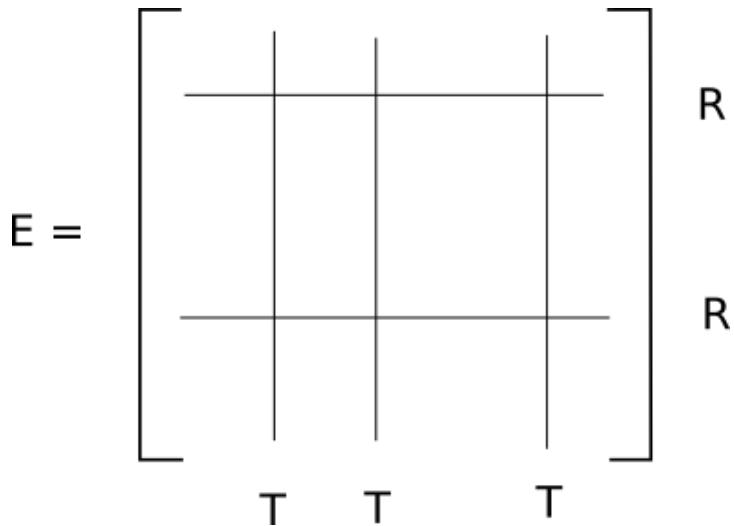
- i) os valores dos v_j serão aumentados
- ii) os valores dos u_i serão reduzidos

Algoritmo

1. Seja G o subgrafo da igualdade (subgrafo induzido pelas arestas tal que $e_{ij}=0$ na matriz de excessos)
2. Encontra um emparelhamento máximo M em G e um subconjunto cobertura $Q \subseteq V$ com $|Q|=|M|$ (cobertura mínima)
3. Se $|M|=n$, então o emparelhamento é perfeito, ou seja, o excesso é zero em toda aresta de M . Retorne M , \vec{u} e \vec{v} . Caso contrário, devemos aumentar M . Para isso, seja $R=Q \cap X$ e $T=Q \cap Y$. Seja ainda:

$$\epsilon = \min\{e_{ij} : x_i \notin R \wedge y_j \notin T\}$$

Isso pode ser realizado utilizando marcadores na matriz de excessos: buscamos entradas que não estejam nas linhas marcadas por R e nas colunas marcadas por T , como ilustra a figura a seguir.



Devemos encontrar o menor número de linhas horizontais e verticais que englobam todos os zeros da matriz de excessos.

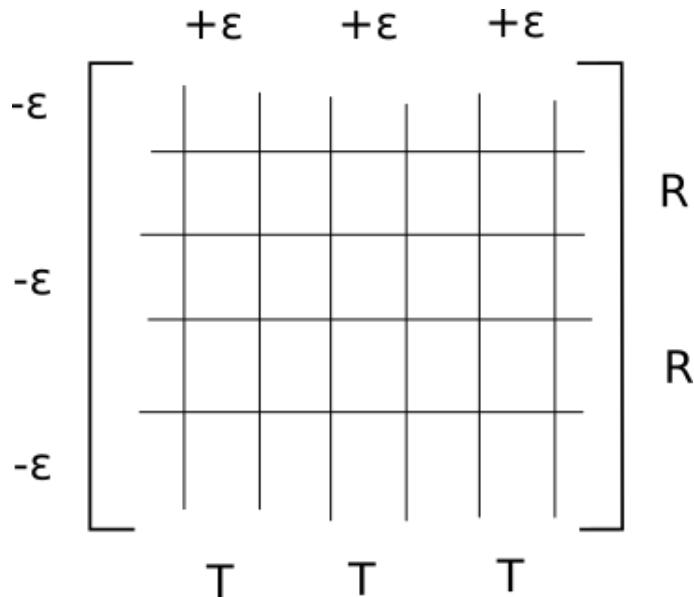
4. Atualize os vetores de pesos \vec{u} e \vec{v} como segue:

- a) Diminua u_i de ϵ se $x_i \notin R$: $u_i = u_i - \epsilon$
- b) Aumente v_j de ϵ se $y_j \in T$: $v_j = v_j + \epsilon$

Essa regra define o seguinte padrão de atualização baseado na posição dos elementos da matriz de excessos, o que é a base para o próximo passo.

5. Atualize a matriz de acessos como:

- a) $x_i \notin R \wedge y_j \notin T : e_{ij} = e_{ij} - \epsilon$
- b) $x_i \notin R \wedge y_j \in T : e_{ij} = e_{ij} - \epsilon + \epsilon = e_{ij}$
- c) $x_i \in R \wedge y_j \in T : e_{ij} = e_{ij} + \epsilon$
- d) $x_i \in R \wedge y_j \notin T : e_{ij} = e_{ij} + \epsilon - \epsilon = e_{ij}$



Em resumo, esse padrão nos diz que só importa o que está fora de R e T e o que está na intersecção de R e T.

A seguir iremos mostrar um exemplo prático da aplicação do algoritmo.

Ex: Considere a seguinte matriz de custos.

$$W = \begin{bmatrix} 3 & 4 & 3 & \boxed{5} & 5 & 2 \\ 5 & \boxed{8} & 4 & 3 & 6 & 1 \\ \boxed{9} & 2 & 1 & 7 & 9 & 4 \\ 3 & 4 & 3 & 5 & 4 & \boxed{6} \\ \boxed{8} & 7 & 2 & 5 & 5 & 3 \\ 3 & 2 & 1 & 4 & \boxed{6} & 3 \end{bmatrix}$$

Encontre um emparelhamento perfeito de custo mínimo.

Primeiro, devemos definir os vetores de pesos \vec{u} e \vec{v} , que nesse caso são:

$$\vec{u} = (5, 8, 9, 6, 8, 6)$$

$$\vec{v} = (0, 0, 0, 0, 0, 0)$$

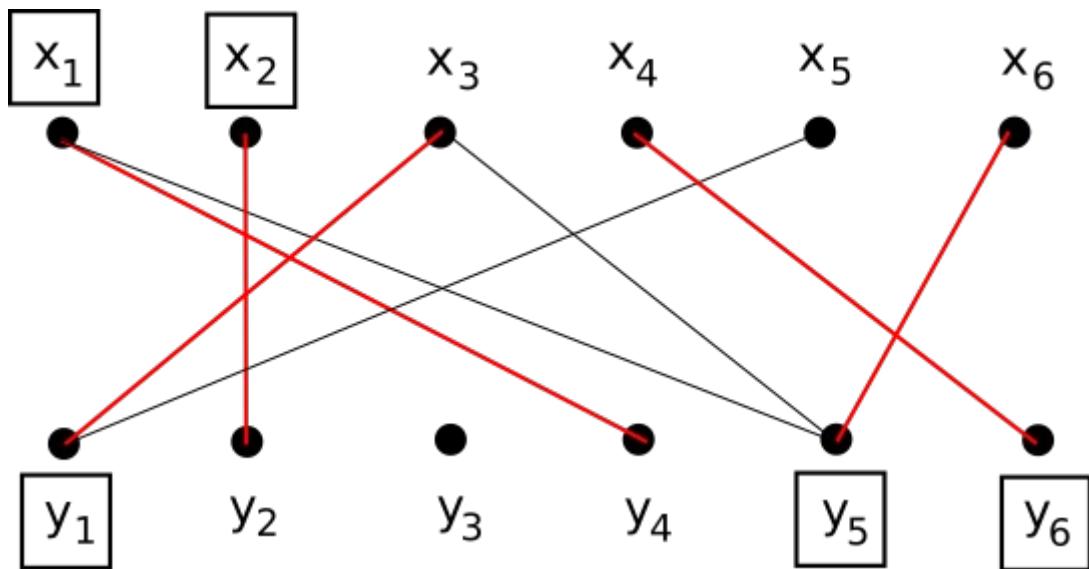
Assim, podemos definir a seguinte matriz de excessos:

$$v = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$u = \begin{matrix} 5 \\ 8 \\ 9 \\ 6 \\ 8 \\ 6 \end{matrix} \begin{bmatrix} 2 & 1 & 2 & 0 & 0 & 3 \\ 3 & 0 & 4 & 5 & 2 & 7 \\ 0 & 7 & 8 & 2 & 0 & 5 \\ 3 & 2 & 3 & 1 & 2 & 0 \\ 0 & 1 & 6 & 3 & 3 & 5 \\ 3 & 4 & 5 & 2 & 0 & 3 \end{bmatrix}$$

$$e_{ij} = u_{ij} + v_{ij} - w_{ij}$$

O subgrafo da equidade possui um emparelhamento M com no máximo 5 arestas, portanto a cobertura mínima terá 5 vértices. A figura a seguir ilustra o emparelhamento e a cobertura obtidos.



$$M = \{(x_1, y_1), (x_2, y_1), (x_4, y_6), (x_5, y_5)\}$$

$$Q = \{x_1, x_2, y_1, y_5, y_6\}$$

Note que temos $|M|=|Q|=5$. Para a obtenção de M e Q , podemos tanto aplicar o algoritmo Húngaro visto em aulas anteriores, como encontrar o número mínimo de linhas horizontais e verticais que cobre todos os zeros da matriz de excesso (5 retas = 5 vértices = 5 arestas). De acordo com o emparelhamento e a cobertura definidos, podemos marcar as duas primeiras linhas (x_1 e x_2) e

a primeira, quinta e sexta colunas (y_1 , y_5 e y_6), gerando o seguinte padrão. Veja que $\epsilon=1$, pois é o menor elemento da região não atingida pelas retas R e T.

	2	1	2	0	0	3	R
	3	0	4	5	2	7	R
	0	7	8	2	0	5	
	3	2	3	1	2	0	
	0	1	6	3	3	5	
	3	4	5	2	0	3	
T				T	T		

Atualizando os vetores de pesos, temos:

$$\vec{u} = (5, 8, 8, 5, 7, 5) \quad (\text{subtrai } \epsilon=1 \text{ dos valores fora de R})$$

$$\vec{v} = (1, 0, 0, 0, 1, 1) \quad (\text{soma } \epsilon=1 \text{ dos valores dentro de T})$$

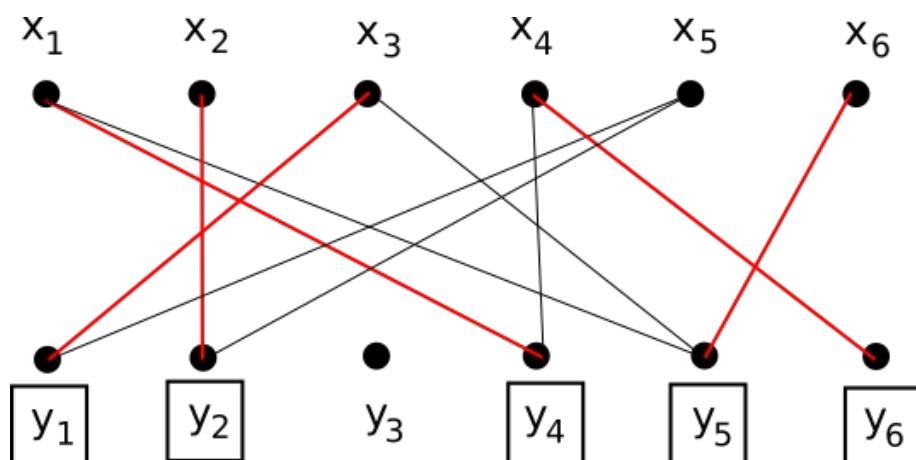
Podemos agora atualizar a matriz de excessos, que fica:

$$v = (1 \ 0 \ 0 \ 0 \ 1 \ 1)$$

$$u = \begin{matrix} 5 \\ 8 \\ 8 \\ 5 \\ 7 \\ 5 \end{matrix} \begin{bmatrix} 3 & 1 & 2 & 0 & 1 & 4 \\ 4 & 0 & 4 & 5 & 3 & 8 \\ 0 & 6 & 7 & 1 & 0 & 5 \\ 3 & 1 & 2 & 0 & 2 & 0 \\ 0 & 0 & 5 & 2 & 3 & 5 \\ 3 & 3 & 4 & 1 & 0 & 3 \end{bmatrix}$$

$$e_{ij} = u_{ij} + v_{ij} - w_{ij}$$

Novamente, o subgrafo da equidade possui um emparelhamento M com 5 arestas, portanto a cobertura mínima terá 5 vértices. A figura a seguir ilustra o emparelhamento e a cobertura obtidos.



De acordo com o emparelhamento e a cobertura definidos, podemos marcar as duas primeiras e as três últimas colunas (y_1 , y_2 , y_4 , y_5 e y_6), gerando o seguinte padrão. Veja que $\epsilon=2$, pois é o menor elemento da região não atingida pelas retas.

3	1	2	0	1	4
4	0	4	5	3	8
0	6	7	1	0	5
3	1	2	0	2	0
0	0	5	2	3	5
3	3	4	1	0	3

T T T T T T

Atualizando os vetores de pesos, temos:

$$\vec{u} = (3, 6, 6, 3, 5, 3) \quad (\text{subtrai } \epsilon=2 \text{ dos valores fora de } R)$$

$$\vec{v} = (3, 2, 0, 2, 3, 3) \quad (\text{soma } \epsilon=2 \text{ dos valores dentro de } T)$$

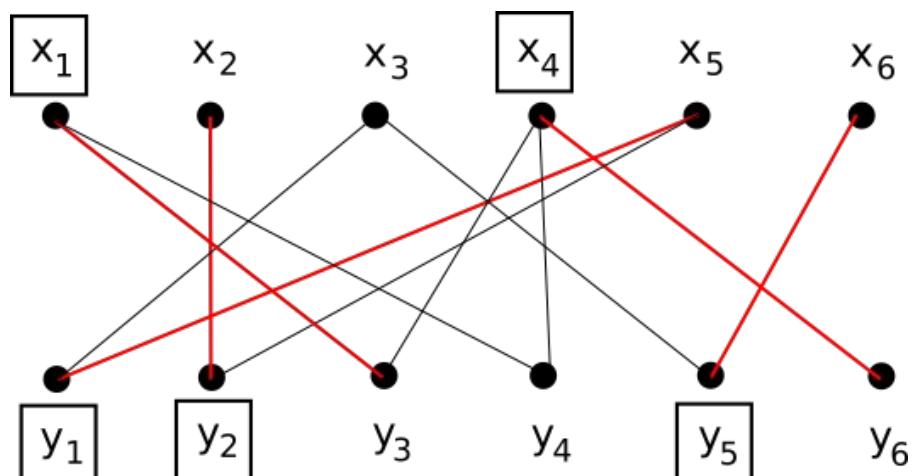
Podemos agora atualizar a matriz de excessos novamente:

$$v = (3 \ 2 \ 0 \ 2 \ 3 \ 3)$$

$$u = \begin{matrix} 3 \\ 6 \\ 6 \\ 3 \\ 5 \\ 3 \end{matrix} \begin{bmatrix} 3 & 1 & 0 & 0 & 1 & 4 \\ 4 & 0 & 2 & 5 & 3 & 8 \\ 0 & 6 & 5 & 1 & 0 & 5 \\ 3 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 2 & 3 & 5 \\ 3 & 3 & 2 & 1 & 0 & 3 \end{bmatrix}$$

$$e_{ij} = u_{ij} + v_{ij} - w_{ij}$$

O subgrafo da equidade possui um emparelhamento M com 5 arestas, portanto a cobertura mínima terá 5 vértices. A figura a seguir ilustra o emparelhamento e a cobertura obtidos.



De acordo com o emparelhamento e a cobertura definidos, podemos marcar as linhas 1 e 4 em R e as colunas 1, 2 e 5 em T. Note que $\epsilon=1$, pois é o menor elemento da região não atingida.

	3	1	0	0	1	4	R
	4	0	2	5	3	8	
	0	6	5	1	0	5	
	3	1	0	0	2	0	R
	0	0	2	2	3	5	
	3	3	2	1	0	3	
T	T			T			

Atualizando os vetores de pesos, temos:

$$\vec{u} = (3, 5, 5, 3, 4, 3) \quad (\text{subtrai } \epsilon=1 \text{ dos valores fora de R})$$

$$\vec{v} = (4, 3, 0, 2, 4, 3) \quad (\text{soma } \epsilon=2 \text{ dos valores dentro de T})$$

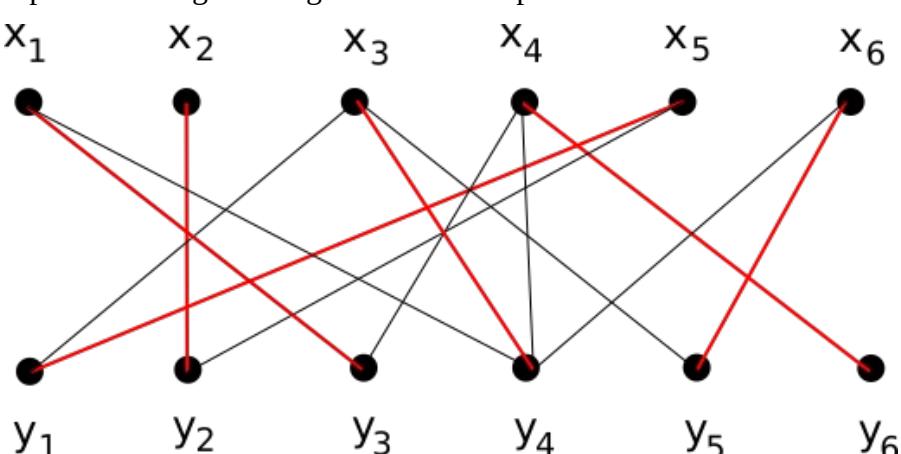
A nova matriz de excessos fica:

$$v = (4 \ 3 \ 0 \ 2 \ 4 \ 3)$$

$$u = \begin{matrix} 3 \\ 5 \\ 5 \\ 3 \\ 4 \\ 2 \end{matrix} \left[\begin{array}{cccccc} 4 & 2 & 0 & 0 & 2 & 4 \\ 4 & 0 & 1 & 4 & 3 & 7 \\ 0 & 6 & 4 & 0 & 0 & 4 \\ 4 & 2 & 0 & 0 & 3 & 0 \\ 0 & 0 & 2 & 1 & 3 & 4 \\ 3 & 3 & 1 & 0 & 0 & 2 \end{array} \right]$$

$$e_{ij} = u_{ij} + v_{ij} - w_{ij}$$

O subgrafo da equidade possui um emparelhamento M com 6 arestas, o que gera um emparelhamento perfeito. A figura a seguir ilustra o emparelhamento e a cobertura obtidos.



Portanto, a solução ótima é dada por:

$$M = \{(x_1, y_3), (x_2, y_2), (x_3, y_4), (x_4, y_6), (x_5, y_1), (x_6, y_5)\}$$

O ganho total de M é:

$$w(M) = \sum_{e \in M} w_{ij} = 3 + 8 + 7 + 6 + 8 + 6 = 18 + 20 = 38$$

Pode-se mostrar que:

$$w(M) = \sum_{i=1}^n u_i + \sum_{j=1}^n v_j = (3+5+5+3+4+2)+(4+3+0+2+4+3)=22+16=38$$

Uma observação importante é que podemos aplicar o algoritmo visto aqui em problemas de minimização. Para isso, defina:

$$K = \max\{w_{ij}\} + 1 \quad (K \text{ é o maior peso somado de uma unidade})$$

e calcule a nova matriz de pesos:

$$W' = K - W$$

Basta então utilizar a nova matriz W' como entrada para o algoritmo.

“Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende.”
(Leonardo da Vinci)

Problemas NP-Completos

Há problemas computacionais que não podem ser resolvidos por algoritmos, nem que dispuséssemos de um período de tempo infinito: são os problemas incomputáveis ou indecidíveis.

Um exemplo clássico, formulado por Alan Turing, considerado um dos pais da computação, é o problema da parada (The halting problem). Dado um programa P e uma entrada I, devemos decidir se o programa terminará sua execução com essa entrada ou será executado para sempre. Não pode existir um algoritmo que receba P e resolva esse problema de decisão!

Veremos a seguir um breve esboço do principal argumento utilizado na prova formal.

Suponha que exista uma função computável $\text{halts}(f)$ que retorna True se a subrotina f termina e False caso contrário (nunca termina). Seja a função g() a seguir:

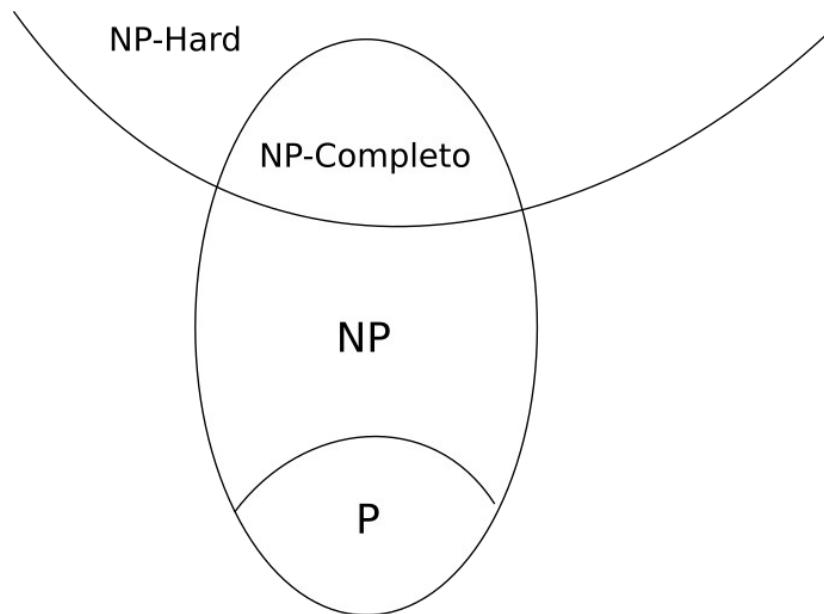
```
def g():
    if halts(g):
        loop_forever
```

Note que se $\text{halts}(g)$ retorna True, significa que ela termina, mas então ela entra em loop infinito, o que gera uma contradição!

Mas se $\text{halts}(g)$ retorna False, então a função g irá terminar (não entra no if), o que também gera uma contradição!

Portanto, a suposição inicial de que existe um função computável $\text{halts}()$ é FALSA!

Dentre os problemas computáveis, podemos organizar a seguinte estrutura de classes: problemas P, NP, NP-Completos e NP-Hard. A figura a seguir ilustra um diagrama.



Def: Dizemos que $p \in P$ se o problema é facilmente solucionável, ou seja, existe um algoritmo A com complexidade $O(n^k)$ (polinomial) que resolve P. Em termos matemáticos, o problema p pode ser resolvido por uma máquina de Turing determinística.

Exemplos são problemas como a ordenação, que possuem algoritmos exatos.

Def: Dizemos que $p \in NP$ se o problema é facilmente verificável, mas não facilmente solucionável. Em outras palavras, não existe algoritmo eficiente que resolva p , mas dada uma solução, é fácil verificar se ela é válida. Em termos matemáticos, pode ser resolvido por uma máquina de Turing não determinística.

Def: Os problemas $p \in NP$ para os quais $\forall q \in NP$ podem ser reduzidos a p de maneira eficiente (em tempo polinomial) são conhecidos como NP-Completos.

NP-Completos são problemas NP-Hard que estão em NP. Por isso são geralmente muito difíceis de serem resolvidos.

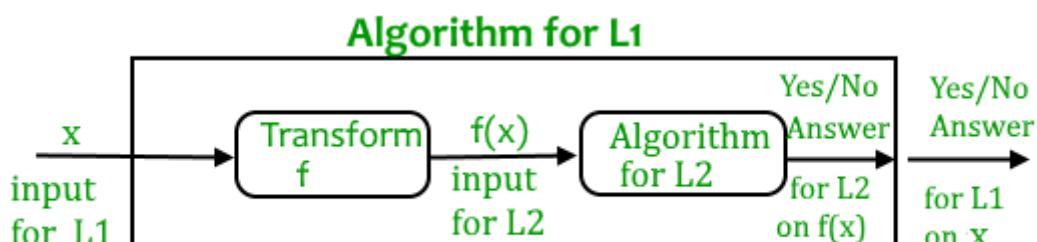
Def: Existem problemas $p \notin NP$ para os quais verificar se uma dada solução é válida sequer é viável: são os problemas NP-Hard (considerados os mais difíceis de todos)

Obs: A diferença é que problemas NP-Completo são problemas de decisão, enquanto que problemas NP-Hard são problemas de otimização.

A pergunta natural que surge é: o que é redução?

Sejam L_1 e L_2 dois problemas de decisão e suponha que o algoritmo A_2 resolva L_2 . Em outras palavras, se y é uma entrada para L_2 , então A_2 retornará True ou False, dependendo se $y \in L_2$ ou $y \notin L_2$.

Ideia: encontrar uma transformação de L_1 para L_2 de modo que o algoritmo A_2 possa ser parte de um algoritmo A_1 para resolver L_1 .



Como provar que um problema p é NP-Completo?

Parece impossível: Todo problema $q \in NP$ deve ser reduzido a p

Ideia: tomar um problema NP-Completo conhecido e reduzi-lo a p . Se a redução for possível em tempo polinomial, prova-se que p é NP-Completo por transitividade da redução, ou seja, se um problema NP-Completo é reduzido a p em tempo polinomial, então todos os problemas em NP são reduzíveis a p em tempo polinomial.

Qual foi o primeiro problema a provado a ser NP-Completo?

SAT (Boolean satisfiability problem)

Consiste em determinar se existe uma interpretação que satisfaz uma dada expressão Booleana arbitrária. Existe alguma forma de atribuir valores lógicos as variáveis Booleanas da expressão lógica de modo que ela seja verdadeira?

Precisamos construir a Tabela-Verdade.

Por exemplo, considere a seguinte expressão lógica:

$$((a \vee \neg b) \wedge (\neg c \vee d)) \rightarrow (\neg(e \rightarrow (\neg f \wedge g)) \vee h)$$

Como temos 8 variáveis lógicas, precisamos de $2^8 = 256$ linhas na Tabela-Verdade.

Claramente, a complexidade do problema é exponencial: $O(2^n)$

Um dos 7 problemas do milênio: P = NP ?

Instituto Clay dos EUA oferece um prêmio de 1 milhão de dólares para quem provar que P = NP.

Traria implicações inimagináveis na computação: se eventualmente você conseguir reduzir um problema NP-Completo para P, todos os problemas de NP virariam P (criptografia, segurança, etc.)

Grafos Hamiltonianos

Iremos iniciar o estudo de problemas NP-Completos com um problema clássico da computação.

Dado um grafo $G = (V, E)$, decidir se G é Hamiltoniano, ou seja, se é possível sair de um vértice v , visitar todo outro vértice de G exatamente uma vez e retornar a v .

Para isso precisamos definir o que são grafos Hamiltonianos e como são caracterizados.

Def: Um ciclo Hamiltoniano é um ciclo que engloba todo vértice de G (ciclo não permite repetição)

Def: Um grafo G é Hamiltoniano se e somente se G possui um ciclo Hamiltoniano (dual de Euler)

Obs: $\forall n > 2$ K_n é Hamiltoniano

Processo de crescimento das arestas

Supor um grafo G arbitrário não Hamiltoniano

$$\begin{array}{ccccccc} G & \rightarrow & G' & \rightarrow & G'' & \rightarrow & G''' \rightarrow \dots \rightarrow K_n \\ & & +e & & +e & & +e \\ & & & & & & +e \end{array}$$

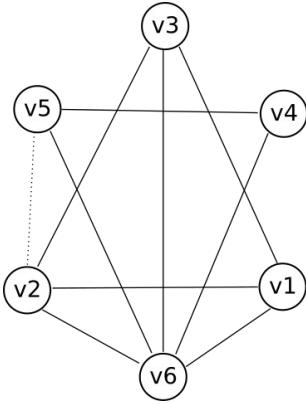
A cada passo, uma aresta é inserida em G . Como K_n é Hamiltoniano, segue que em algum momento entre G e K_n , o grafo torna-se Hamiltoniano. Usaremos essa noção para a definição a seguir.

Def: Um grafo G é não Hamiltoniano maximal se G não é Hamiltoniano mas a adição de qualquer nova aresta o torna Hamiltoniano (está na iminência de se tornar Hamiltoniano)

Ex: considere o grafo G a seguir. Note que G não admite um ciclo Hamiltoniano mas ao adicionar qualquer aresta nova, G torna-se Hamiltoniano

No caso da aresta (v_2, v_5) temos: $v_6 - v_4 - v_5 - v_2 - v_1 - v_3 - v_6$

No caso da aresta (v_1, v_4) temos: $v_6 - v_5 - v_4 - v_1 - v_2 - v_3 - v_6$



Pergunta: Como decidir se um dado grafo G é Hamiltoniano? Problema difícil. (NP-Completo)
Porque? Melhor resultado que se tem até hoje

Teorema (Dirac, 1952)

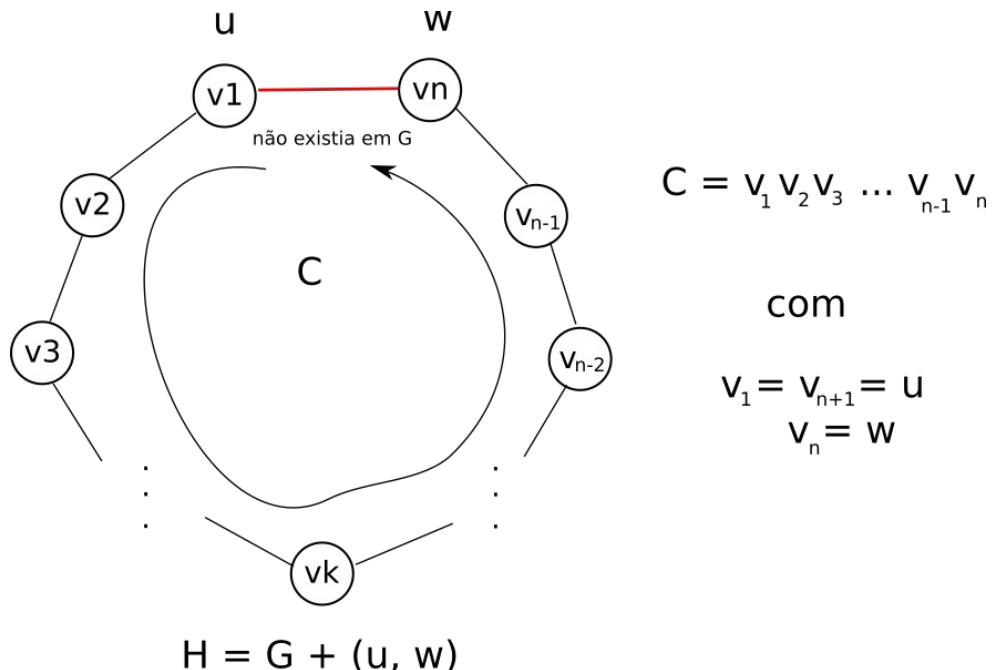
Seja $G = (V, E)$ um grafo básico simples com $|V| = n > 2$. Então,

$$\forall v \in V (d(v) \geq \frac{n}{2}) \rightarrow G \text{ é Hamiltoniano}$$

Em outras palavras, o que esse resultado nos diz é que se cada um dos vértices de G se liga pelo menos a metade dos demais, então G é Hamiltoniano. (esse resultado ainda é considerado o estado da arte na identificação de grafos Hamiltonianos, no sentido que não há nada mais poderoso)

Prova: (por contradição)

Ideia é verificar que não pode existir G que satisfaça propriedade H e não seja Hamiltoniano.
Suponha que $\exists G = (V, E)$ básico simples que satisfaz propriedade mas não é Hamiltoniano.
Considere G como sendo não Hamiltoniano maximal. Então $\exists u, w \in V$ não adjacentes.
Faça $H = G + (u, w)$. H é Hamiltoniano, então \exists ciclo C que passa por $\forall v \in V$
Chamaremos $u = v1$ e $w = vn$



Defina 2 conjuntos de vértices, S e T, como segue:

$$S = \{ v_i : \exists \text{ aresta que liga } u \text{ a } v_{i+1} \}$$

$$T = \{ v_j : \exists \text{ aresta que liga } w \text{ a } v_j \}$$

Obs: Quem está em S? Não importa! Quantos elementos estão em S? Isso importa

Assim, temos que $|T| = d(w)$ e $|S| = d(u)$ (número de ligações).

Agora, iremos verificar que \exists ao menos um vértice que não está em S nem em T: v_n

Note que:

i) $v_n \notin S$: Porque? \exists aresta que liga u a $v_{n+1} = u$? Não pois não há loops!

ii) $v_n \notin T$: Porque? \exists aresta que liga w a $v_n = w$? Não pois não há loops!

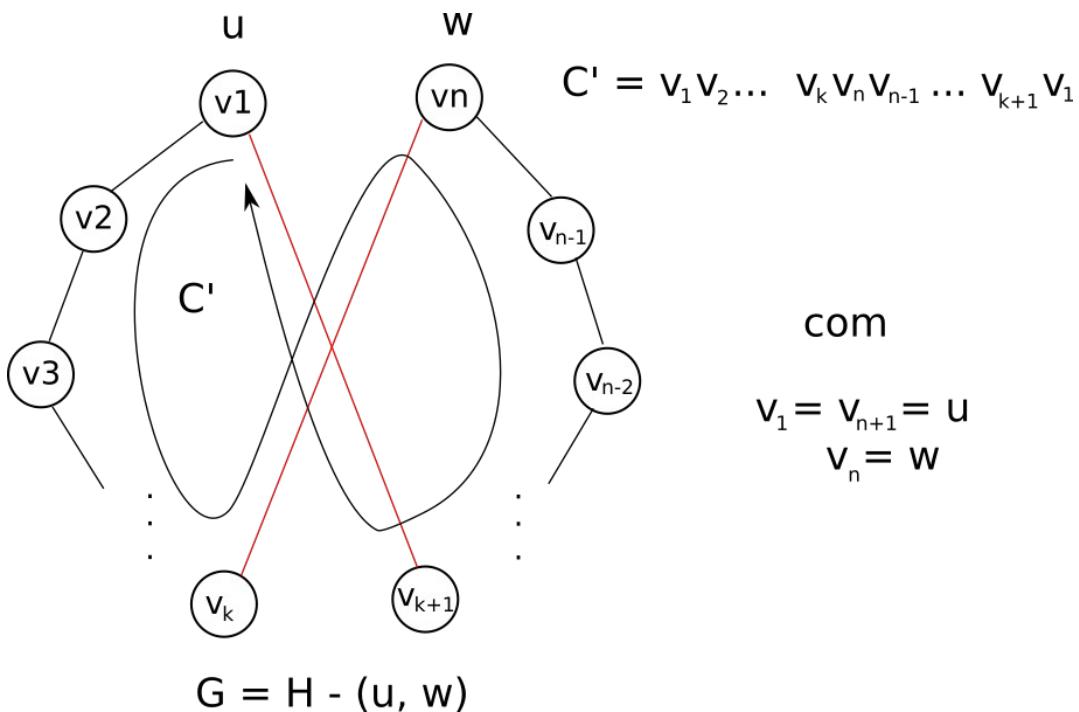
Dessa forma, $v_n \notin S \cup T$, o que implica que $|S \cup T| < n$

O próximo passo consiste em responder a pergunta: $\exists v_k \in S \cap T$?

Suponha que sim, se chegarmos numa contradição é porque não pode existir. Vamos considerar que $v_k \in S \cap T$. Então,

i) se $v_k \in S$: \exists aresta que liga u a v_{k+1}

ii) se $v_k \in T$: \exists aresta que liga w a v_k



$$C' = v_1 v_2 \dots v_k v_n v_{n-1} \dots v_{k+1} v_1$$

com

$$\begin{aligned} v_1 &= v_{n+1} = u \\ v_n &= w \end{aligned}$$

Porém, nesse caso G seria Hamiltoniano pois existiria um ciclo C' que envolve todo vértice de G. Isso fere a definição inicial de que G é não Hamiltoniano Maximal, gerando uma contradição e portanto invalidando a suposição de que $\exists v_k \in S \cap T$. O fato é que $\nexists v_k \in S \cap T$. Isso implica que $S \cap T = \emptyset$.

Dessa forma, da teoria dos conjuntos temos:

$$|S \cup T| = |S| + |T| - |S \cap T| = d(u) + d(w)$$

Mas como de (*) temos que $|S \cup T| < n$ finalmente chegamos a:

$$d(u) + d(w) < n$$

o que é uma contradição para a primeira suposição, pois todo grafo que satisfaz H tem

$$d(u) \geq \frac{n}{2} \quad \text{e} \quad d(w) \geq \frac{n}{2}$$

para quaisquer u e w em V, ou seja, $d(u) + d(w) \geq n$. Em outras palavras, a hipótese de que existe G que satisfaz H e não seja Hamiltoniano é falsa.

Conclusão: Não existe grafo G que satisfaz a propriedade H e não seja Hamiltoniano.
(se H é verdade então é certeza que G é Hamiltoniano)

Mas isso garante que é simples decidir se um dado G é Hamiltoniano? Não, pois posso ter grafos G para os quais H falha e mesmo assim eles são Hamiltonianos. Exemplo: grafo 2-regular conexo

Problema em aberto: ninguém nunca demonstrou a volta. Nem mesmo propôs um critério mais efetivo (do tipo se e somente se)

O teorema anterior, apesar de poderoso, não nos fornece uma caracterização completa de grafos Hamiltonianos. Basicamente, ele diz que todo grafo que satisfaz a propriedade H é Hamiltoniano, mas nem todo grafo Hamiltoniano satisfaz a propriedade H.

Decidir se um grafo $G = (V, E)$ é Hamiltoniano é um problema NP-Completo.

O problema do caixeiro-viajante

Dado um grafo $G = (V, E, w)$ encontrar um ciclo Hamiltoniano de custo mínimo, ou seja, obter o ciclo C que minimiza:

$$w(C) = \sum_{e \in C} w(e)$$

Descrição do problema: A partir de um vértice inicial v_0 , visitar todos os demais uma única vez, voltando para v_0 , caminhando o mínimo possível. É um problema de otimização e não de decisão (NP-Hard).

Pergunta: como obter um ciclo Hamiltoniano de custo mínimo?

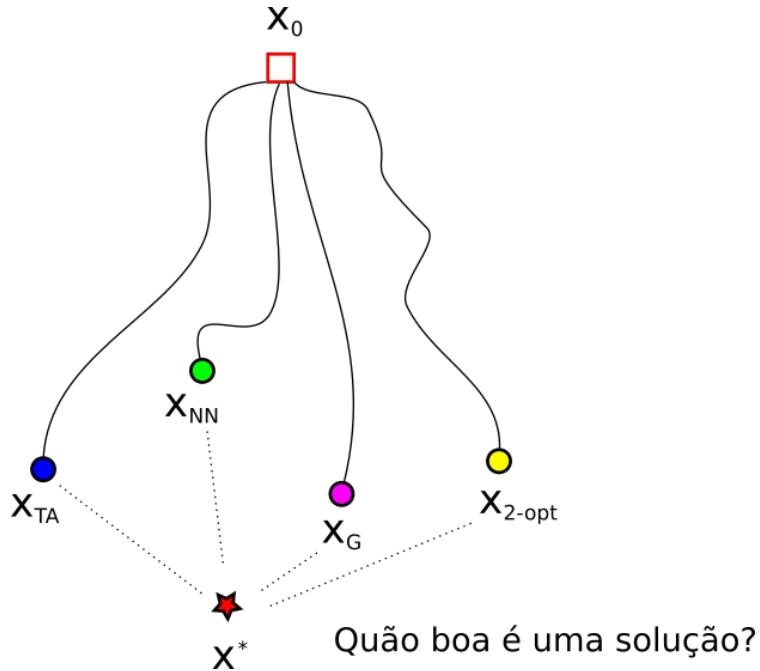
- Não se conhece algoritmo ótimo (não há garantias de que termos a melhor solução x^*)
- Devemos trabalhar com algoritmos aproximados.

Mas como medir a qualidade da solução obtida por um algoritmo aproximado?

Devemos medir quão próximo da solução ótima é a solução aproximada. Definem-se

$$\begin{aligned} f(x^*) &: \text{custo da solução ótima (melhor possível)} \\ f(x_A) &: \text{custo da solução obtida pelo algoritmo A} \end{aligned}$$

Dizemos que o algoritmo A fornece uma c -aproximação para o TSP se $f(x_A) \leq c f(x^*)$



TSP métrico

Seja $G = (V, E, w)$ um grafo conexo com $w: E \rightarrow R^+$, em que w é uma função de distância. Então, a desigualdade triangular é satisfeita, ou seja:

$$\forall i, j, k \quad w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$$

Isso significa que tomar atalhos é sempre vantajoso (corta caminhos).

Uma instância do TSP em que os pesos das arestas do grafo de entrada G é uma função de distância é conhecida como TSP métrico.

Algoritmos para o TSP

A seguir apresentamos alguns dos principais algoritmos aproximados para o TSP.

Nearest Neighbor

O algoritmo mais simples para o TSP utiliza uma abordagem gulosa: visitar sempre o vértice mais próximo do vértice atual. Porém, esse algoritmo não fornece boas soluções, em especial quando o número de vértices cresce de maneira arbitrária.

```

Nearest_Neighbor(G, w, v0) {
    H = [v0]
    while |H| < n {
        x = Last(H)      // returns the last element of H
        Define vi as the nearest neighbor of x
        InsertEnd(H, vi) // inserts vi at the end of H
    }
    InsertEnd(H, v0)
}

```

Teorema: A solução obtida pelo algoritmo Nearest Neighbor no TSP métrico satisfaz:

$$f(x_{NN}) \leq \frac{1}{2}(\log_2 n + 1)f(x^*)$$

Note que se o número de vértices do grafo G aumenta, a qualidade da solução cai de maneira logarítmica.

Análise da complexidade

1. Note que o loop WHILE executa n vezes
2. Obter o último elemento da lista H pode ser realizado em tempo constante $O(1)$ se mantivermos um contador para o número de elementos em H .
3. Para definir o vizinho mais próximo de x , é preciso verificar todas as distâncias da linha de x na matriz de adjacências, o que é $O(n)$
4. Inserção no final da lista H também pode ser feito em $O(1)$ se mantivermos um contador para o número de elementos de H

Custo total: $n \times (O(1) + O(n) + O(1)) = O(n^2)$

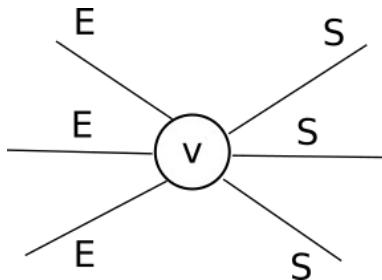
Twice-Around

Antes de apresentarmos o algoritmo Twice-Around, precisamos definir o que é um grafo Euleriano.

Def: Um grafo $G = (V, E)$ é Euleriano se G contém um circuito Euleriano, que é uma trilha fechada que envolve toda aresta de G . Em outras palavras, a partir de um vértice v , é possível passar uma única vez por cada aresta de G e voltar a v .

Teorema: Um grafo $G = (V, E)$ é Euleriano, se e somente se, $\forall v \in V (d(v) \% 2 = 0)$, ou seja, se o grau de todos os vértices é par.

Em um grafo Euleriano é possível extrair um circuito Euleriano, ou seja, atravessar toda aresta exatamente uma única vez, voltando ao vértice inicial. A ideia é que para cada aresta de entrada, haverá uma outra aresta de saída, de modo que não é possível ficar preso durante o circuito.



Para entender como podemos extrair um circuito Euleriano a partir de um grafo em que todos os graus são pares, apresentamos o algoritmo de Hierholzer.

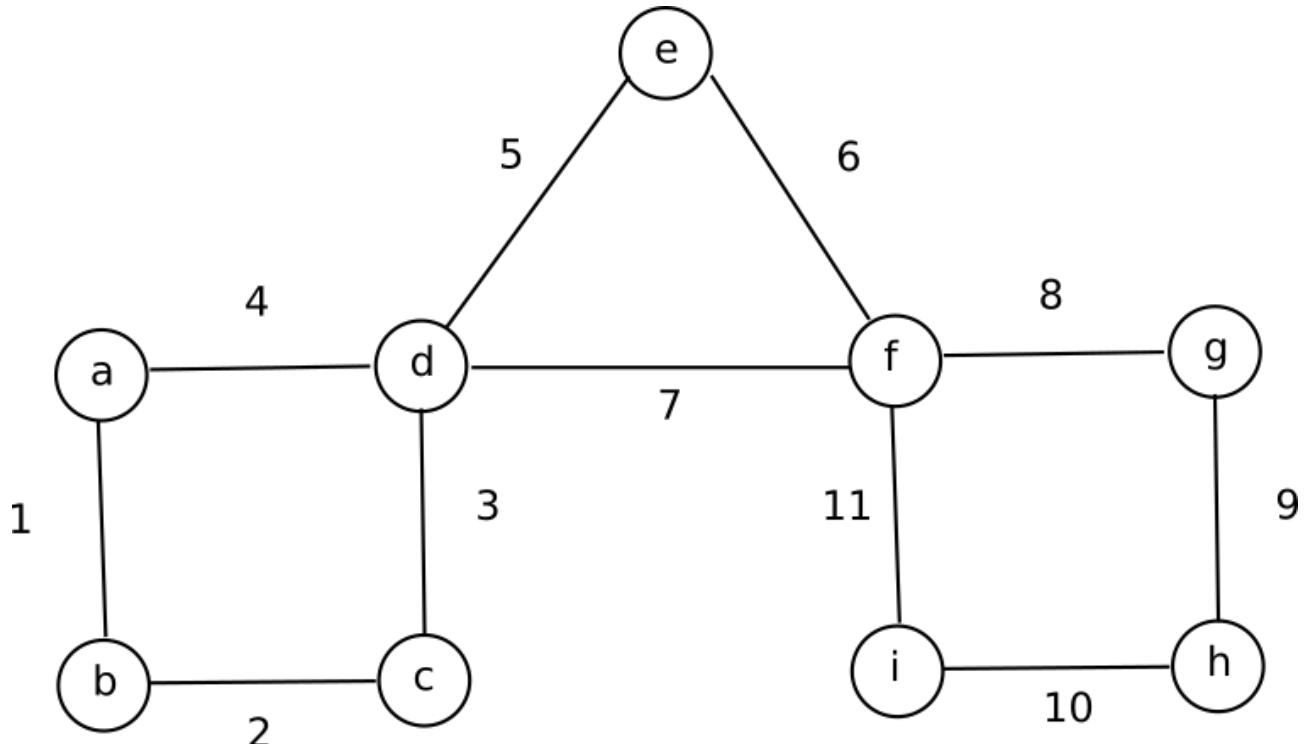
```
Eulerian_Circuit(G, v) {
    // Areças iniciam desmarcadas
    for each e = (u,v) in E
        α(u,v) = False
    S = Stack()
    Q = Queue()
```

```

push(S, v)
while S ≠ ø {
    u = Peek(S)      // verifica quem é o vértice do topo
    dead_end = True
    for each v ∈ N(u) {
        if α(u,v) == False { // pode atravessar
            push(S, v)
            α(u,v) = True   // já passou pela aresta
            dead_end = False
            break
        }
    }
    if dead_end {
        u = pop(S)
        InsertLeft(Q, u)
    }
}
return Q
}

```

Vejamos um exemplo ilustrativo a seguir.



Iremos iniciar no vértice **a** e adicionar as arestas na ordem em que aparecem na figura:

$S = [a]$
 $Q = \emptyset$

Após inserirmos os vértices b, c, d e a no topo de S, chegamos num beco sem saída:

$S = [a, b, c, d, a]$ // desempilha a e adiciona em Q
 $Q = \emptyset$

Devemos desempilhar a e adicionar a esquerda em Q:

$$S = [a, b, c, d]$$

$$Q = [a]$$

Seguindo a ordem, iremos adicionar e, f e d no topo de S, até chegarmos em outro beco sem saída:

$$S = [a, b, c, d, e, f, d]$$

$$Q = [a]$$

Devemos desempilhar d e adicionar a esquerda em Q

$$S = [a, b, c, d, e, f]$$

$$Q = [d, a]$$

Seguindo a ordem, adicionamos os vértices g, h, i e f, novamente atingindo um beco sem saída:

$$S = [a, b, c, d, e, f, g, h, i, f]$$

$$Q = [d, a]$$

Como todas as arestas já foram marcadas, só nos resta desempilhar os vértices de S em Q:

$$S = \emptyset$$

$$Q = [a, b, c, d, e, f, g, h, i, f, d, a]$$

Note que no final do algoritmo a pilha encontra-se vazia e a fila contém o circuito Euleriano.

Análise da complexidade

1. Para marcar inicialmente as arestas o custo é $O(m)$
2. O loop WHILE executa uma vez para cada vértice, ou seja n vezes.
3. Verificamos quem é o vértice do topo da pilha n vezes: Peek é $O(1) \times n = O(n)$
4. Como cada aresta é visitada uma única vez, o FOR executa $d(u)$ vezes, em que $d(u)$ é o grau de u .
5. Como sabemos, pelo Handshaking Lema, temos que:

$$\sum_{i=1}^n d(v_i) = 2m$$

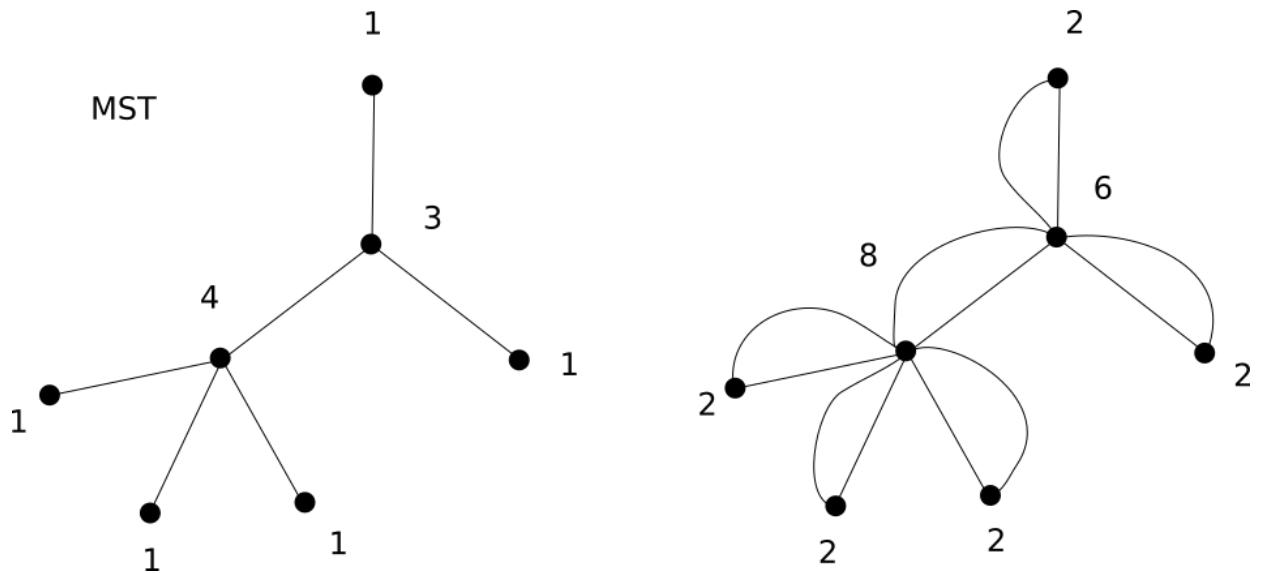
ou seja, o somatório dos graus dos vértices de um grafo G é igual a duas vezes o número de arestas.

6. Portanto, o custo total do algoritmo é dado por:

$$O(m) + O(n) + O(2m) = O(m)$$

uma vez que em grafos conexos $m > n - 1$, ou seja, m domina n .

Teorema: Seja $G = (V, E)$ um grafo conexo. Se T é uma MST de G , então se definirmos um supergrafo T' duplicando toda aresta de T , T' será Euleriano.



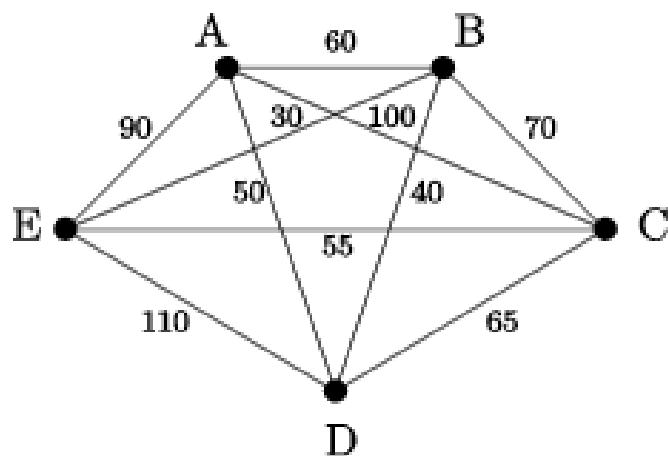
É fácil notar que ao duplicar todas as arestas da MST, estamos multiplicando os graus por 2. Assim, os nós folhas, passarão a ter grau 2 e todos os demais nós internos serão múltiplos de 2 e portanto pares. A seguir apresentamos o algoritmo Twice_Around, conhecido também como Double-Tree, para encontrar uma solução aproximada para o TSP métrico.

```

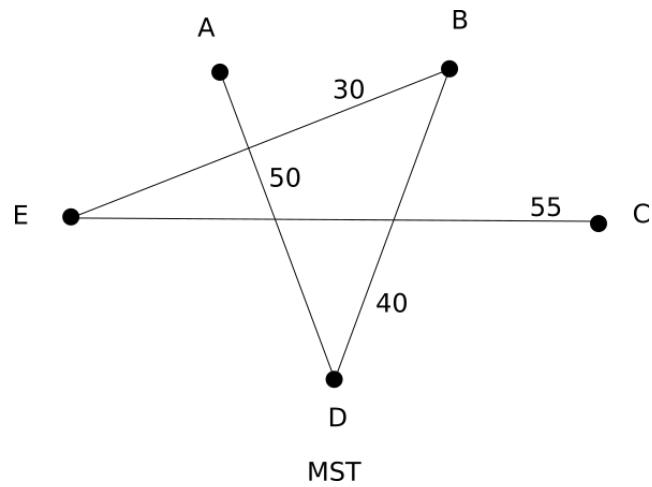
Twice_Around(G, w, v0) {
    H = ∅
    T = MST(G)
    for each e ∈ T
        T = T ∪ {e}
    L = Eulerian_Circuit(T)
    while L ≠ ∅ {
        l = RemoveFirst(L)
        if l ∉ H
            H = H ∪ {l}
    }
    return H
}

```

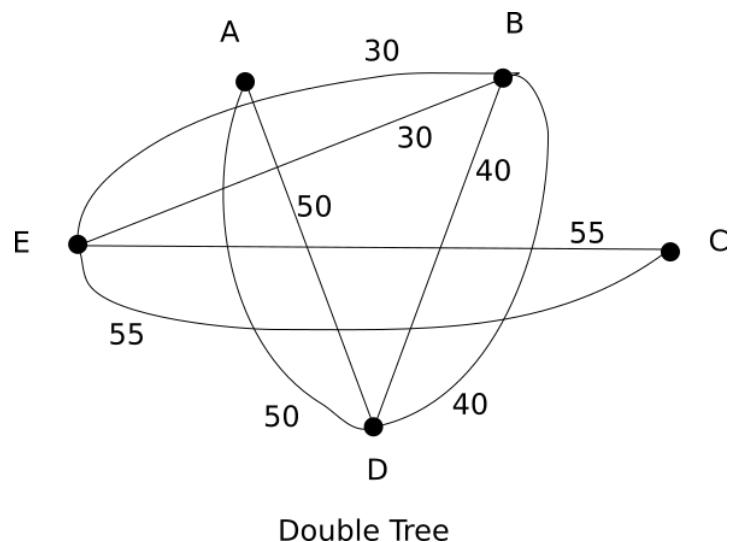
Veremos a seguir um exemplo ilustrativo.



1º passo: Extrair MST de G



2º passo: Duplicar arestas da MST



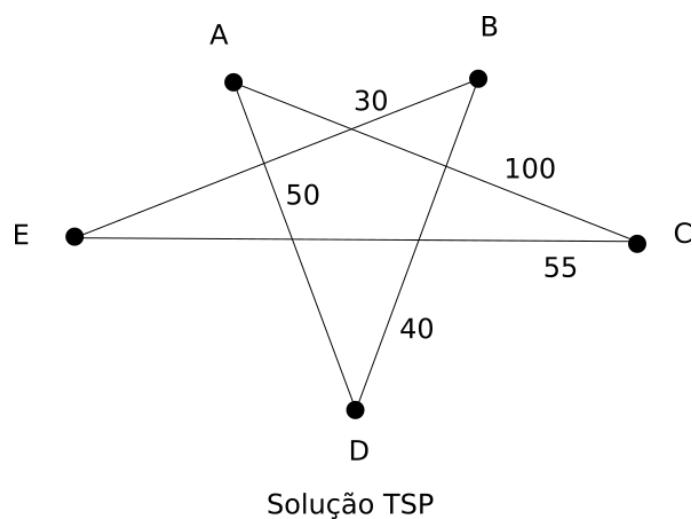
Double Tree

3º passo: Extrair um circuito Euleriano L

$$L = [A, D, B, E, C, E, B, D, A]$$

4º passo: Eliminar as repetições (equivale a tomar atalhos)

$$H = [A, D, B, E, C, A]$$



Solução TSP

Note que o custo do ciclo Hamiltoniano obtido é $w(C) = 50 + 40 + 30 + 55 + 100 = 275$

Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (estruturas estáticas) ou $O(m \log n)$ (estruturas dinâmicas).
2. A duplicação das arestas da MST pode ser feita em $O(m)$
3. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$
4. Por fim, o WHILE é executado uma vez para cada aresta duplicada: $2m$ iterações, o que é $O(m)$
5. Portanto, o custo final é: $O(m \log n) + O(m) + O(m) + O(m) = O(m \log n)$

Ou seja, o crítico no algoritmo Twice-Around é o cálculo da MST.

Veremos a seguir um resultado muito importante sobre a qualidade da solução obtida pelo algoritmo Twice-Around.

Teorema: A solução obtida pelo algoritmo Twice-Around no TSP métrico satisfaz:

$$f(x_{TA}) \leq 2f(x^*)$$

ou seja, a solução do TA é no máximo 2 vezes pior que a solução ótima (não importa o número de vértices de G). Melhor que Nearest Neighbor.

É fácil verificar que se x^* é a solução ótima, então se subtrairmos uma aresta e de x^* , obtemos uma árvore T, ou seja, $T = x^* - e$. Também sabemos que essa árvore T tem peso maior ou igual que a MST do grafo e por isso temos:

$$f(x^*) > w(T) \geq w(T_{MST})$$

o custo de x^* é maior que de T pois o ciclo tem uma aresta a mais

Então temos que:

$$f(x^*) > w(T_{MST})$$

o que implica em

$$2f(x^*) > 2w(T_{MST}) \quad (*) \quad (\text{é o peso do tour de Euler extraído no passo 2})$$

Se G é Euclidiano (tomar atalhos é sempre mais vantajoso devido a desigualdade triangular)

$$f(x_{TA}) \leq 2w(T_{MST}) \quad (**)$$

E portanto, combinando (*) e (**)

$$2f(x^*) > 2w(T_{MST}) \geq f(x_{TA})$$

O algoritmo de Christofides

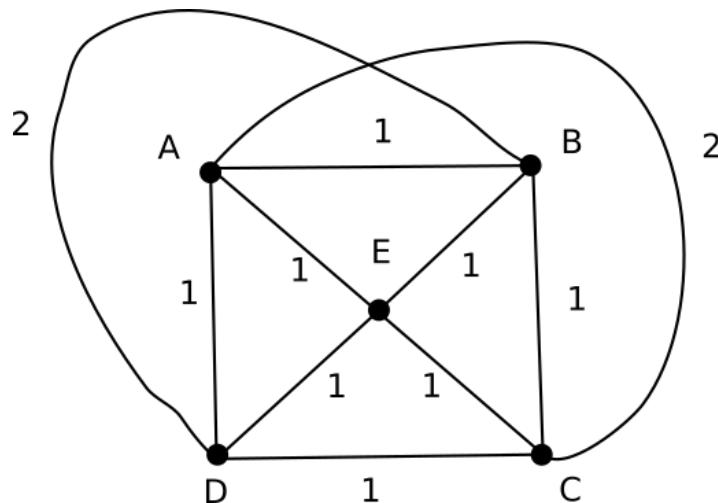
A ideia do algoritmo de Christofides consistem em melhorar ainda mais o algoritmo Twice-Around, modificando a forma com que transformamos a MST de G em um grafo Euleriano. Ao invés de simplesmente duplica as arestas da árvore, devemos encontrar um emparelhamento perfeito de custo mínimo entre os vértices de grau ímpar na MST. Dessa forma, com menos duplicações de arestas, o custo adicional é menor.

```

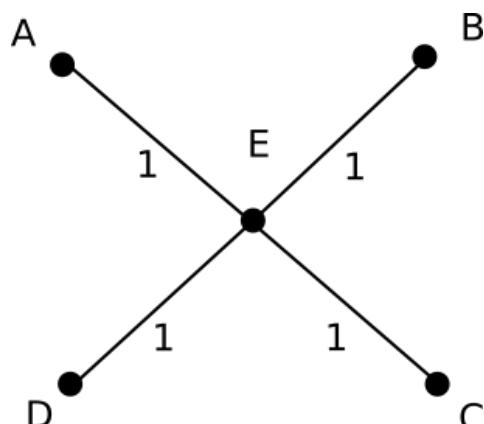
Christofides(G, w, v0) {
    H = ∅
    T = MST(G)                                // Extrai MST de G
    K = Odd_Vertices(T)                      // Constrói Kn (ímpares)
    M = Minimum_Weight_Matching(Ku)        // Emparelhamento mínimo
    T = T + M                                  // Adiciona arestas de M
    T = Eulerian_Circuit(T)                   // Obtém circuito Euleriano
    while L ≠ ∅ {
        l = RemoveFirst(L)
        if l ∉ H                               // Remove as repetições
            H = H ∪ {l}
    }
    return H
}

```

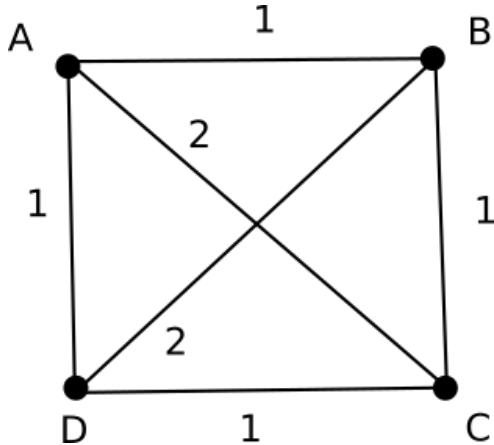
A seguir veremos um exemplo ilustrativo do funcionamento do algoritmo de Christofides. Considere o seguinte grafo G.



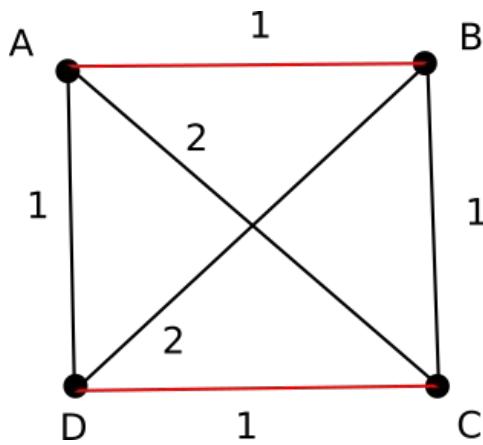
1º passo: Extrair a MST de G



2º passo: criar grafo K_n com vértices de grau ímpar e ponderar as arestas com os caminhos mínimos entre cada par de vértices.



3º passo: encontrar um emparelhamento de custo mínimo entre os vértices de grau ímpar.



Pode-se mostrar que o número de emparelhamentos perfeitos no K_n é dado por:

$$\prod_{i \text{ ímpar}, i < n} i$$

No caso de $n = 4$, temos $1 \times 3 = 3$

No caso de $n = 6$, temos $1 \times 3 \times 5 = 15$

No caso de $n = 8$, temos $1 \times 3 \times 5 \times 7 = 105$

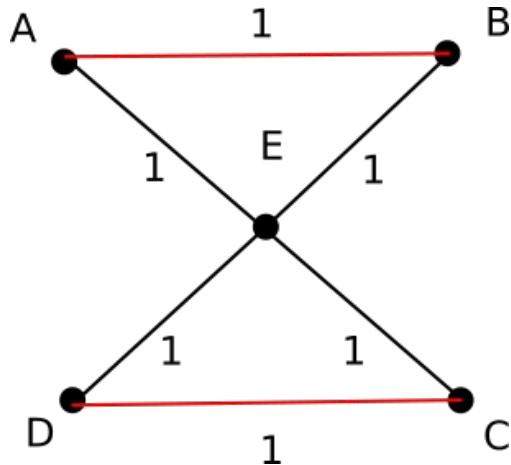
No caso de $n = 10$, temos $1 \times 3 \times 5 \times 7 \times 9 = 945$

Note que o número de possíveis emparelhamentos perfeitos cresce exponencialmente, o que inviabiliza a busca exaustiva pelo emparelhamento mínimo. Existe um algoritmo com complexidade $O(n^2 m)$ que obtém o emparelhamento M de custo mínimo em um grafo arbitrário (não precisa ser bipartido). É o algoritmo de Edmonds, também conhecido como *Blossom algorithm*. Para os leitores interessados, indicamos o link a seguir:

https://en.wikipedia.org/wiki/Blossom_algorithm

Basicamente, trata-se de uma generalização do método que vimos para resolver o problema da alocação ótima. A dificuldade adicional aqui é que em grafos não bipartidos, podem ocorrer ciclos de comprimento ímpar, o que gera alguns padrões que necessitam de um tratamento especial.

4º passo: Adicione as arestas de M à MST



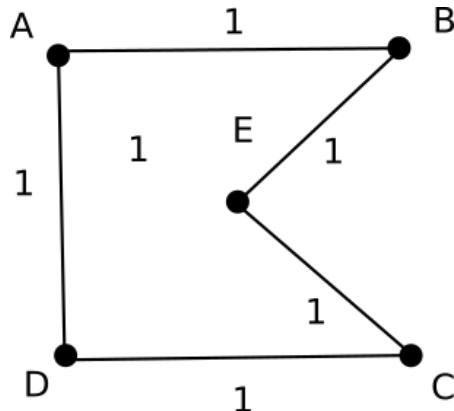
Agora, todos os vértices possuem grau par, então é possível extrair um circuito Euleriano.

5º passo: Encontre um circuito Euleriano L

$$L = [A, B, E, C, D, E, A]$$

6º passo: Eliminar as repetições (equivalente a tomar atalhos)

$$L = [A, B, E, C, D, A]$$



Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (estruturas estáticas) ou $O(m \log n)$ (estruturas dinâmicas).

2. Obter o emparelhamento de custo mínimo M entre os vértices de grau ímpar: seja z o número de vértices ímpares. O grafo completo formado pelos vértices ímpares, denotado por K_z , terá m' arestas, onde:

$$m' = \frac{z(z-1)}{2}$$

3. Para ponderar os pesos do grafo K_z , será preciso encontrar caminhos mínimos entre cada par de vértices. Podemos executar k vezes o algoritmo de Dijkstra ou 1 vez o algoritmo de Floyd-Warshall. Em ambos os casos, temos uma complexidade $O(k^3)$

4. Para a construção do emparelhamento M de mínimo custo, o algoritmo de Edmonds possui complexidade $O(k^2 m')$

5. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$

6. Por fim, o WHILE é executado uma vez para cada aresta duplicada: $2m$ iterações, o que é $O(m)$

7. Portanto, o custo final é:

$$O(m \log n) + O(k^3) + O(k^2 m') + O(m)$$

Ou seja, se tivermos poucos vértices de grau ímpar, o custo é dominado pela construção da MST, mas quando o número de vértices ímpares é grande, ou seja, $k \rightarrow n$, o custo é dominado pela construção do grafo K_z e obtenção do emparelhamento de custo mínimo M.

A seguir, iremos demonstrar um resultado fundamental sobre o algoritmo de Christofides.

Teorema: O algoritmo de Christofides fornece uma $3/2$ -aproximação para o TSP métrico.

Prova: Desejamos mostrar que:

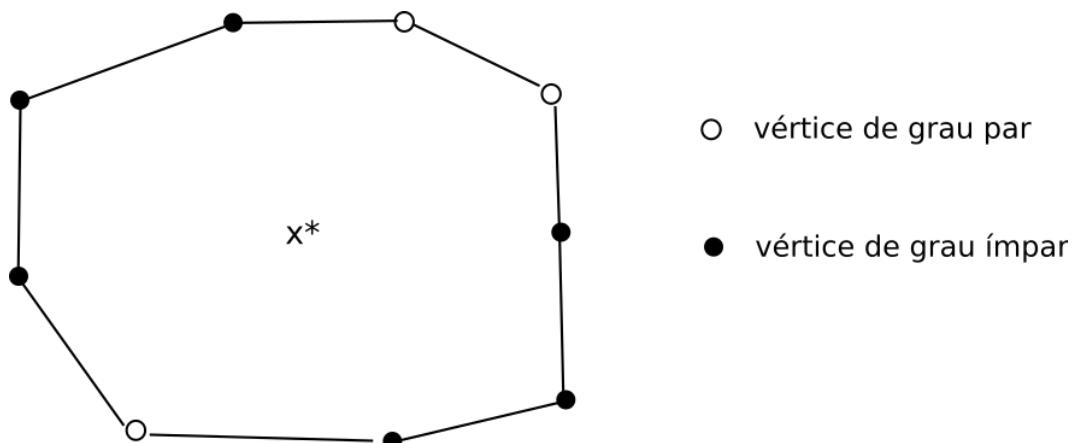
$$f(x_c) \leq \frac{3}{2} f(x^*)$$

onde X^* é a solução ótima.

Sabemos que o custo da MST T é menor que x^* , ou seja:

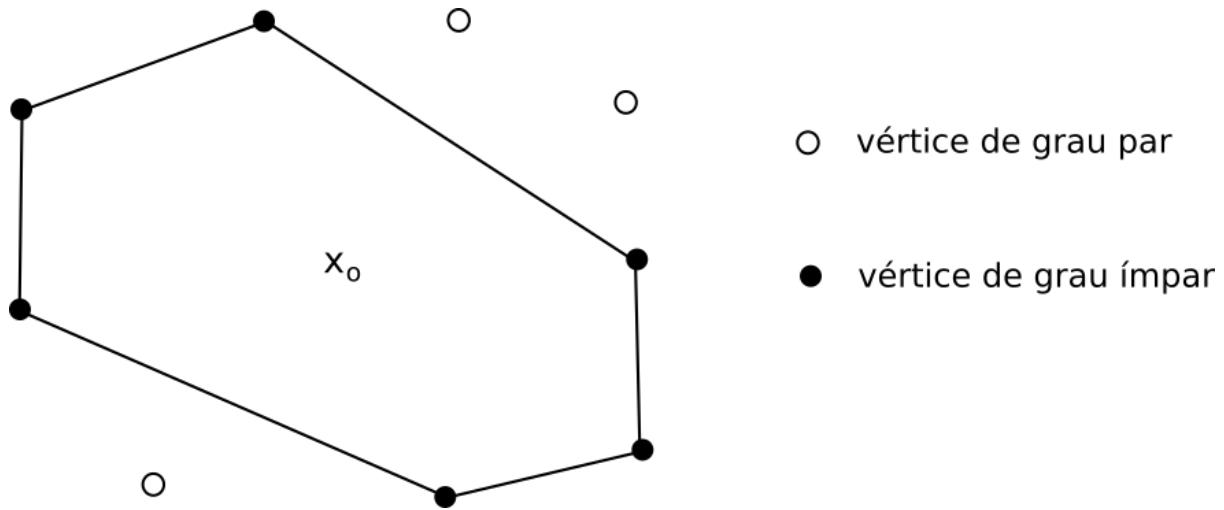
$$w(T_{MST}) < f(x^*)$$

pois $x^* - e$ é uma árvore com peso maior ou igual ao peso da MST de G. Basta agora provar que o custo do emparelhamento M entre os vértices ímpares é menor ou igual a $\frac{1}{2} f(x^*)$. Lembre-se que aqui, ao invés de duplicar todas as arestas da MST, adicionamos menos arestas a MST. Iniciamos a discussão com o ciclo Hamiltoniano de custo mínimo, cujo custo é $f(x^*)$.



onde os vértices pretos denotam os vértices de grau ímpar (subconjunto O de odd degree) e os vértices brancos denotam os vértices de grau par. Existe um ciclo Hamiltoniano de custo mínimo

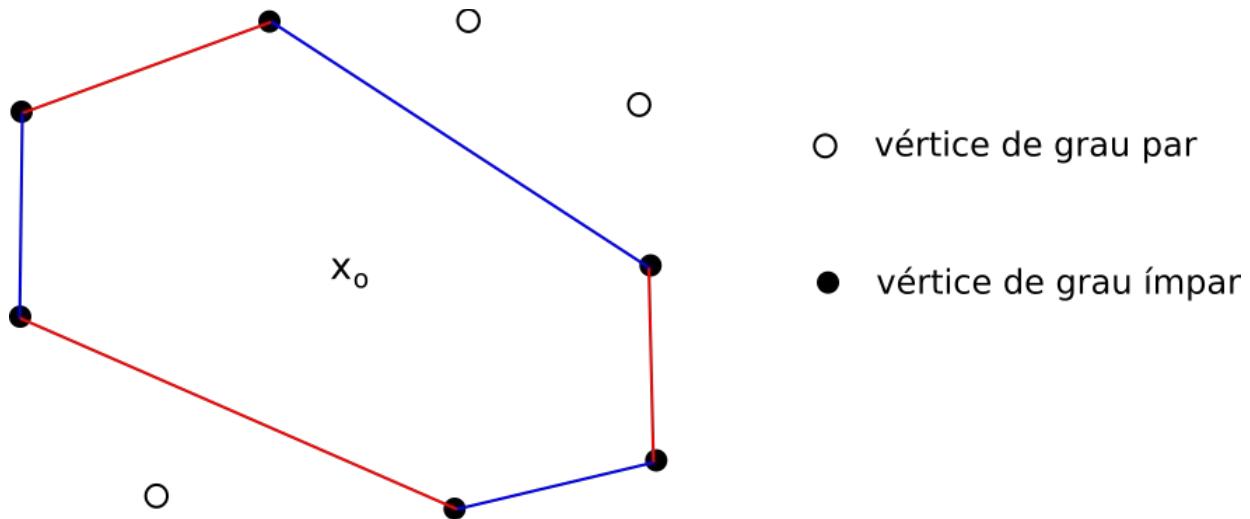
que passa apenas pelos vértices do subconjunto O , denotado por x_o , conforme ilustra a figura a seguir:



O custo do ciclo x_o é menor que o custo da solução ótima, pois ele passa por menos vértices:

$$f(x_o) \leq f(x^*)$$

Porém, note que podemos particionar o ciclo x_o em arestas alternadas de duas cores: vermelhas e azuis. Note que como o número de vértices ímpares é sempre par, o número de arestas no ciclo x_o será sempre par, e assim, temos 2 emparelhamentos distintos com o mesmo número de arestas: M' (azuis) e M'' (vermelhas).



Como o peso total dos 2 emparelhamentos é igual ao custo do ciclo x_o , temos:

$$w(M') + w(M'') = f(x_o) \leq f(x^*)$$

Se os pesos de M' e M'' forem iguais:

$$w(M') \leq \frac{f(x^*)}{2}$$

Se os pesos de M' e M'' forem diferentes, seja $\bar{M} = \min\{M', M''\}$. Então, temos:

$$w(\bar{M}) \leq \frac{f(x^*)}{2}$$

Assim, como no algoritmo de Christofides encontramos o emparelhamento M de custo mínimo entre os vértices de grau ímpar, $M = \bar{M}$ e $w(M) \leq \frac{f(x^*)}{2}$.

Portanto o custo da solução final satisfaz:

$$f(x_c) = w(T_{MST}) + w(M)$$

Como $w(T_{MST}) < f(x^*)$ e $w(M) \leq \frac{f(x^*)}{2}$, temos finalmente:

$$f(x_c) \leq f(x^*) + \frac{f(x^*)}{2} = \frac{3}{2}f(x^*)$$

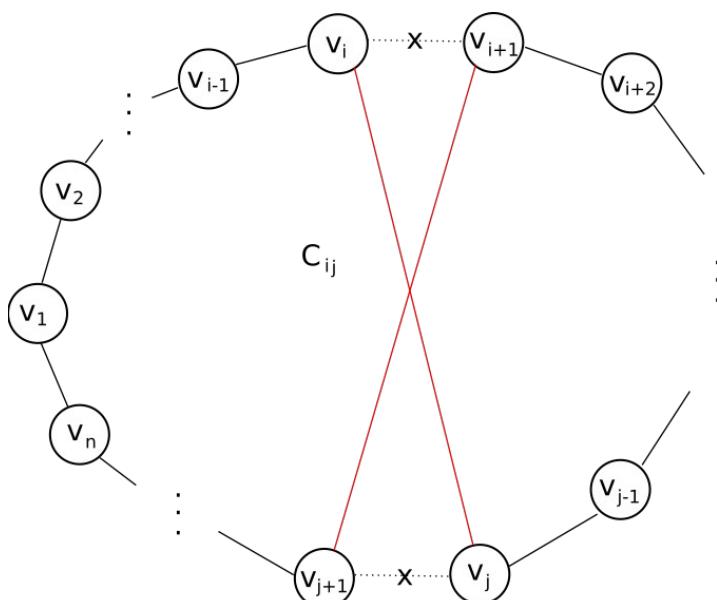
O algoritmo 2-optimal

Ideia: partir de uma inicialização arbitrária e tentar melhorá-la iterativamente a partir de buscas na vizinhança local.

Pseudocódigo

Sem perda de generalidade, seja $C = v_1 v_2 v_3 \dots v_{n-1} v_n$ a solução inicial. Definiremos a operação Two_Opt_Swap(C) conforme a seguir:

```
Two-Opt-Swap( $C$ ,  $i$ ,  $j$ ) {
    Let  $C_{ij}$  be the cycle obtained by rewiring  $v_i$  and  $v_j$  as
     $C' = v_1 v_2 \dots v_i v_j v_{j-1} v_{j-2} \dots v_{i+1} v_{j+1} \dots v_n v_1$ 
```

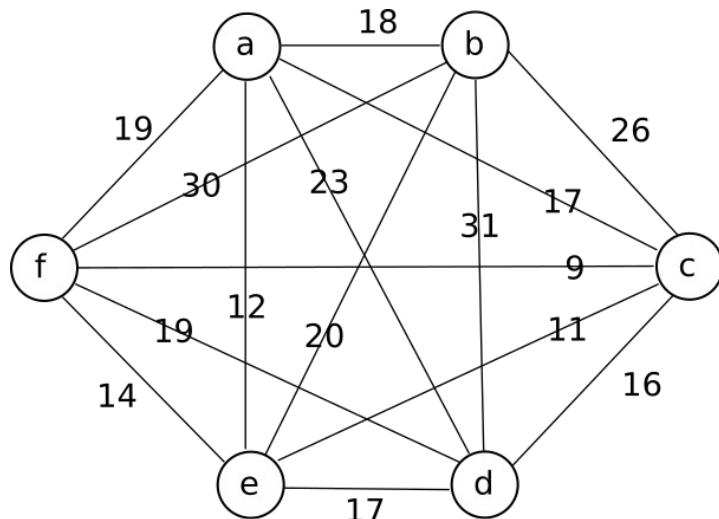


```
} return  $C'$ 
```

Essa operação é a base para a busca local e define como perturbamos a solução corrente na busca por uma solução melhor que esteja nas proximidades. A seguir é mostrado o algoritmo 2-Optimal, também conhecido como 2-Opt.

```
Two-Opt(C, n) {
    best = C
    improved = True
    while improved {
        improved = False
        for i = 1 to n-2 {
            for j = i+2 to n {
                C = Two-Opt-Swap(C, i, j)
                if w(C) < w(best) {
                    best = C
                    improved = True
                }
            }
        }
        C = best
    }
}
```

A seguir mostramos um exemplo ilustrativo da aplicação do algoritmo 2-Opt

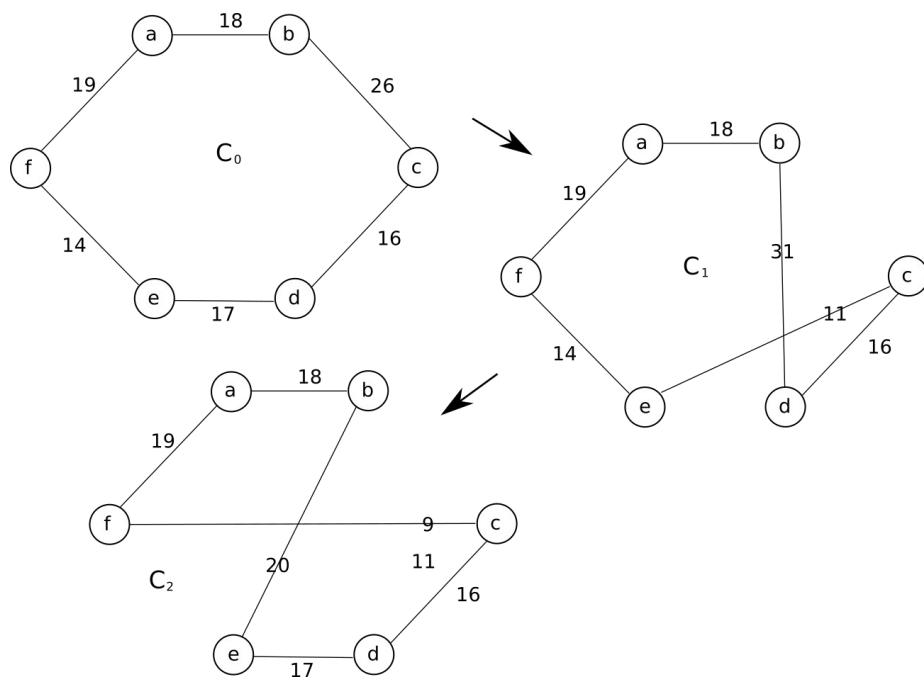


Consideramos como solução inicial o ciclo $C = a \ b \ c \ d \ e \ f \ a$ que possui custo $w(C) = 110$.

C								W
i	j	1	2	3	4	5	6	
1	3	a	c	b	d	e	f	110
1	4	a	d	c	b	e	f	124
1	5	a	e	d	c	b	f	118
1	6	a	f	e	d	c	b	120
2	4	a	b	d	c	e	f	109 *

1	3	a	d	b	c	e	f	a	124
1	4	a	c	d	b	e	f	a	117
1	5	a	e	c	d	b	f	a	119
1	6	a	f	e	c	d	b	a	109
2	4	a	b	c	d	e	f	a	110
2	5	a	b	e	c	d	f	a	103 *
1	3	a	e	b	c	d	f	a	112
1	4	a	c	e	b	d	f	a	117
1	5	a	d	c	e	b	f	a	119
1	6	a	f	d	c	e	b	a	103
2	4	a	b	c	e	d	f	a	110
2	5	a	b	d	c	e	f	a	109
2	6	a	b	f	d	c	e	a	104
3	5	a	b	e	d	c	f	a	99 *
1	3	a	e	b	d	c	f	a	107
1	4	a	d	e	b	c	f	a	114
1	5	a	c	d	e	b	f	a	119
1	6	a	f	c	d	e	b	a	99
2	4	a	b	d	e	c	f	a	105
2	5	a	b	c	d	e	f	a	110
2	6	a	b	f	c	d	e	a	102
3	5	a	b	e	c	d	f	a	103
3	6	a	b	e	f	c	d	a	100
4	6	a	b	e	d	f	c	a	100

PARE!



Análise da complexidade

É possível realizar a operação Two_Opt_Swap(C, i, j) com complexidade O(1). Sendo assim, em uma rodada do algoritmo 2-Opt, temos o seguinte número de operações:

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+2}^n 1 = (n-2) + (n-3) + (n-4) + (n-(n-1)) = (n+n+n+\dots+n) - (2+3+4+\dots+(n-1))$$

Note que o primeiro somatório é justamente $n(n-2)$. Já o segundo somatório é igual a:

$$\left(\sum_{i=1}^{n-1} i \right) - 1 = \frac{n(n-1)}{2} - 1 = \frac{n^2 - n - 2}{2}$$

Logo, temos:

$$T(n) = n^2 - 2n - \left(\frac{n^2}{2} - \frac{n}{2} - 1 \right) = \frac{n^2}{2} - \frac{3n}{2} + 1 = \frac{n^2 - 3n + 2}{2}$$

o que é $O(n^2)$. Portanto, supondo que o loop WHILE mais externo execute K vezes, teremos uma complexidade $O(Kn^2)$. Como $K \ll n$ (K é uma constante muito menor que n), podemos dizer que a complexidade do algoritmo 2-Opt é $O(n^2)$.

Teorema: A solução obtida pelo algoritmo 2-Opt em uma instância do TSP métrico satisfaz:

$$f(x_{2-opt}) \leq (\log n) f(x^*)$$

Iremos omitir a prova deste resultado, mas de qualquer forma, note que a qualidade da solução depende do número de vértices do grafo. Quanto mais vértices o grafo possui, menos garantias de que estamos obtendo soluções próximas à solução ótima.

Uma estratégia comum consiste em utilizar os algoritmos Nearest Neighbor, Twice-Around e Christofides para gerar soluções iniciais para o 2-Opt, na tentativa de melhorá-las ao menos um pouco.

Existem algoritmos exatos para o TSP: um deles é o método da força bruta, que testa todas as combinações possíveis de ciclos de comprimento n e possui complexidade $O(n!)$, sendo inviável para a maioria dos problemas. O algoritmo de Held-Karp utiliza uma abordagem baseada em programação dinâmica para resolver o TSP e possui complexidade $O(2^n n^2)$ (exponencial), o que ainda é inviável para grafos com um grande número de vértices. Por essa razão, em geral, utiliza-se os algoritmos aproximados na solução de problemas reais.

Coloração de vértices

Ideia: particionar o conjunto de vértices V em conjuntos independentes.

Def: Seja $G = (V, E)$ um grafo e P_1, P_2, \dots, P_n uma partição de V. Dizemos de P_i é um subconjunto independente se quaisquer 2 vértices de P_i não compartilham arestas (não são vizinhos).

Objetivo: Particionar o conjunto V no menor número possível de subconjuntos independentes

Def: Uma k-coloração de G atribui uma de K cores a cada vértice de G, de modo que a vértices adjacentes sempre são atribuídas cores/rótulos diferentes

P1: subconjunto dos vértices de cor 1

P2: subconjunto dos vértices de cor 2

...

Pk: subconjunto dos vértices de cor k

Pergunta: Dado um grafo $G = (V, E)$, \exists uma k-coloração para G (com $k > 2$) ? NP-Completo
Em outras palavras queremos responder se dado G é possível encontrar 2, 3, 4, etc. conjuntos independentes.

Problema de fácil resolução: Que tipos de grafos admitem uma 2-coloração?

Def: O número mínimo k para o qual existe uma coloração de G é o número cromático de G, denotado por $\chi(G)$ (é um atributo do grafo).

Propriedades: (nos ajudam a limitar os valores de $\chi(G)$ em problemas reais: limites inferiores e superiores)

a) Se $|V| = n$, então $\chi(G) \leq n$

i) Se G é o K_n , então $\chi(G) = n$

ii) Se G contém K_m como subgrafo, então $\chi(G) \geq m$

Teorema: G é bipartido $\Leftrightarrow \chi(G) = 2$

Pergunta: quanto vale $\chi(K_{3791,7583})$?

Teorema: Seja $G = (V, E)$ e $\Delta(G) = \max\{d(v) : v \in V\}$ (grau máximo). Então, $\chi(G) \leq \Delta(G) + 1$

Prova:

1. Inicie colorindo os vértices em uma sequência arbitrária.

2. Um vértice deve sempre receber uma cor livre, ou seja, diferente da cor dos vizinhos.

3. Seja o grau máximo de um vértice v em G, $\Delta(G)$.

4. Pelo princípio da casa dos pombos, sempre irá existir uma cor livre no conjunto $\{1, 2, 3, \dots, \Delta(G) + 1\}$.

Teorema: Seja $G = (V, E)$ um grafo conexo com $\Delta(G) \geq 3$. Se G não é completo, $\chi(G) \leq \Delta(G)$

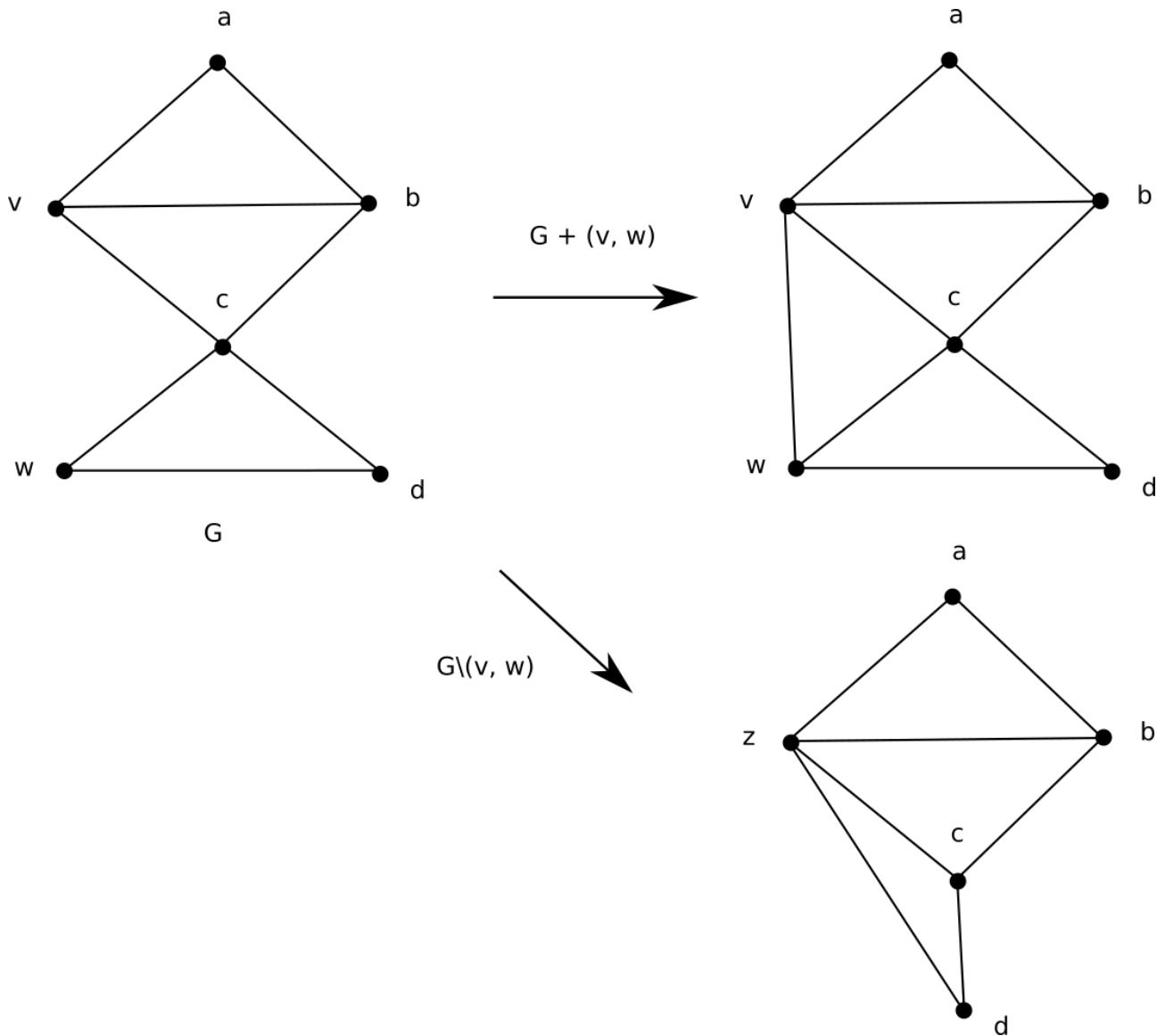
A seguir veremos como podemos definir um algoritmo exato para encontrar o número cromático de um grafo G.

Seja $G = (V, E)$ o grafo de entrada. Se $G = K_n$ então $\chi(G) = n$. Caso contrário, $\exists v, w \in V$ não adjacentes. A ideia é realizar duas alterações estruturais em G:

1. $\alpha_{v,w}(G) = G + (v, w)$ - adição da aresta (v, w) em G

2. $\beta_{v,w}(G) = G \setminus (v, w)$ - condensação dos vértices v e w

onde $G \setminus (v, w)$ é o grafo resultante da fusão dos vértices v e w e posterior eliminação de loops e arestas paralelas que venham a surgir. A figura a seguir ilustra um exemplo.



O método exato de coloração de vértices inicia verificando se $G = K_n$. Em caso positivo, $\chi(G) = n$. Caso contrário, determina-se $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$. O número cromático $\chi(G)$ será o mínimo entre os números cromáticos de $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$, que devem ser calculados recursivamente. Através da recursão, chegaremos em um grafo completo, onde $\chi(G)$ é facilmente computado (caso base). O resultado a seguir fundamenta o algoritmo exato em questão.

Teorema: Seja $G = (V, E)$ um grafo não completo e $v, w \in V$ um par de vértices não adjacentes. Então, o número cromático de G satisfaz:

$$\chi(G) = \min\{\chi(\alpha_{v,w}(G)), \chi(\beta_{v,w}(G))\}$$

Prova:

1. Suponha uma coloração ótima C de G .
2. Se os vértices v, w possuem cores distintas em C , então a aresta (v, w) pode ser adicionada em G sem alterar C . Neste caso, $\chi(G) = \chi(\alpha_{v,w}(G))$.
3. Se os vértices v, w possuem cores iguais a $c_i \in C$, então, ao fundirmos v, w em um único vértice z e atribuirmos a cor c_i a z mantendo as demais cores, temos $\chi(G) = \chi(\beta_{v,w}(G))$.
4. Portanto, $\chi(G)$ deve ser o menor valor entre $\chi(\alpha_{v,w}(G))$ e $\chi(\beta_{v,w}(G))$.

Note que $\chi(G)$ é igual ao número de vértices do menor grafo completo obtido através das operações $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$.

O método a seguir define um algoritmo exato para o problema da coloração de vértices.

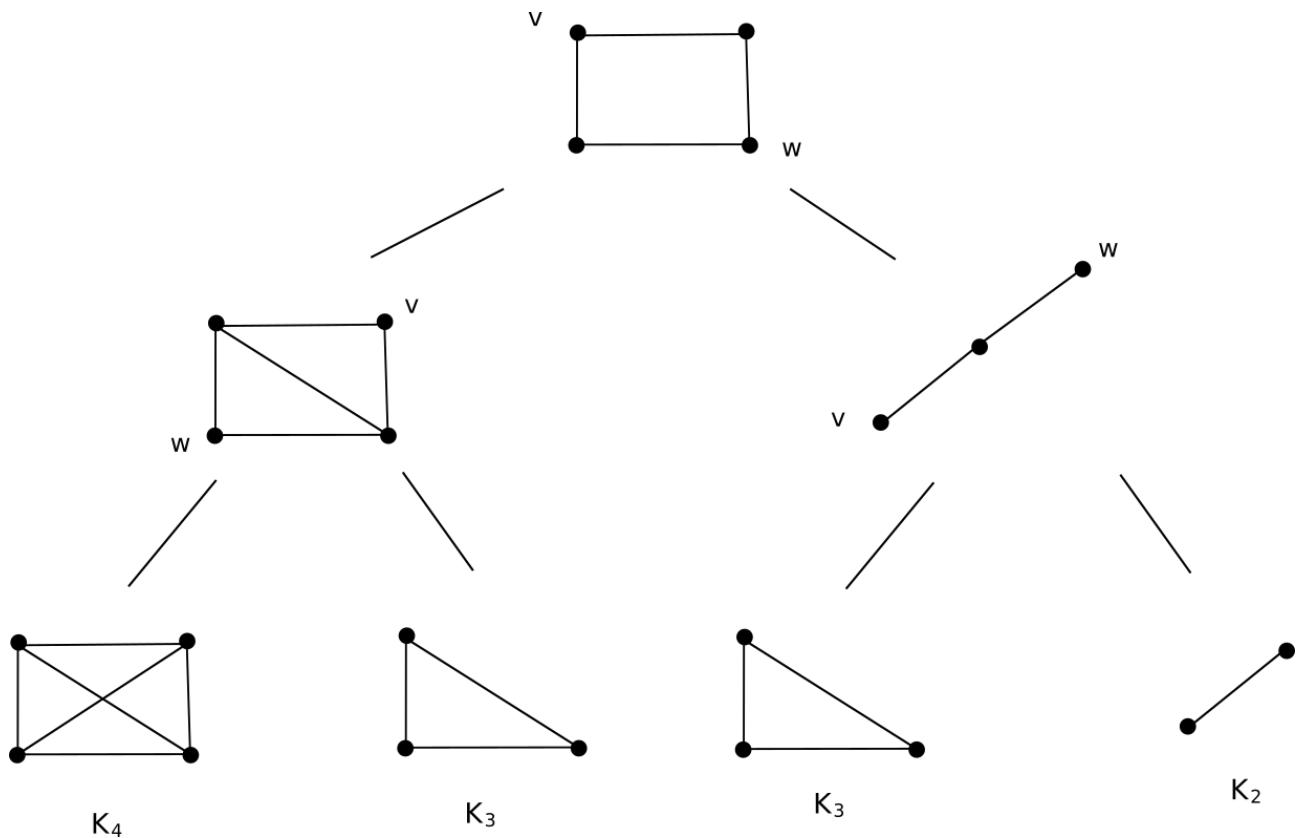
```

Chromatic_Number(G) {
    n = |V|
    if G == Kn
        χ = min{χ, n}
    else {
        Find a pair of non-adjacent neighbors v, w in G
        Chromatic_Number(αv,w(G))
        Chromatic_Number(βv,w(G))
    }
}

χ = n
Chromatic_Number(G)

```

O algoritmo acima dispara um processo recursivo que constrói uma árvore binária em que cada nó corresponde a um grafo: se G é um nó corrente, seu filho a esquerda é $\alpha_{v,w}(G)$ e seu filho a direita é $\beta_{v,w}(G)$. As folhas da árvore, que indicam a condição de parada da recursão, serão sempre grafos completos. A figura a seguir ilustra um exemplo simples.



Portanto, o número cromático de G é $\chi(G)=2$.

Análise da complexidade

Note que o número máximo de grafos a serem examinados é igual ao número de nós total na árvore binária. Como em cada nível ligamos/fundimos apenas 2 vértices, a profundidade da árvore é proporcional ao número de arestas no grafo complementar de G , denotado por m' .

$$2^0 + 2^1 + 2^2 + \dots + 2^{m'} = \sum_{i=0}^{m'} 2^i$$

Note que $2^{i+1} = 2 \cdot 2^i = 2^i + 2^i$, o que nos leva a $2^i = 2^{i+1} - 2^i$. Dessa forma temos:

$$\sum_{i=0}^{m'} (2^{i+1} - 2^i) = 2^{m'+1} - 2^0 = 2^{m'+1} - 1$$

ou seja, a complexidade é exponencial, $O(2^{m'})$. Portanto, esse algoritmo, torna-se inviável para grafos com muito vértices. A seguir iremos discutir um algoritmo aproximado que utiliza uma estratégia gulosa para encontrar o número cromático de G .

Do ponto de vista da classificação dos problemas em computação, encontrar o número cromático $\chi(G)$ de grafos arbitrários é um problema NP-Hard. Alguns fatos relevantes:

1. Não se conhece algoritmo polinomial que colore um grafo arbitrário com exatamente $\chi(G)$ cores.
2. Existe um algoritmo polinomial para coloração de grafos que colore qualquer grafo G com no máximo $O\left(\frac{n}{\log n}\right)\chi(G)$ cores (um pouco acima do valor mínimo).

Abordagem gulosa para coloração de vértices

Suponha que os vértices estejam ordenados em ordem decrescente dos graus, ou seja, temos v_1, v_2, \dots, v_n e seus respectivos graus $\Delta = d_1 \geq d_2 \geq \dots \geq d_n$

Teorema: $\chi(G) = \max_{1 \leq i \leq n} \min \{d_i + 1, i\}$

Prova:

1. No instante em que colorimos v_i (i -ésimo vértice da sequência), note que:
 - a) No máximo $i - 1$ cores foram utilizadas em seus vizinhos, pois antes dele foram exatamente $i - 1$ vértices.
 - b) Mas também, no máximo d_i cores são utilizadas pelos vizinhos, uma vez que o grau de v_i é justamente d_i .

Esse resultado fundamenta o algoritmo de Welsh & Powell, utilizado para colorir um grafo e aproximar o verdadeiro valor do número cromático de G , e que será apresentado a seguir.

```

Welsh_Powell(G) {
    Sort the vertices in decreasing order of degrees
    for each  $v_i \in V$ 
         $C_i = [1, 2, \dots, i]$  // Lista das cores do vértice  $v_i$ 
        for  $i = 1$  to  $n$  {
            color =  $C_i[1]$  // Escolhe a primeira cor da lista
             $v_i.c = color$  // Colore  $v_i$  com a cor escolhida
            for each  $v_j$  in  $N(v_i)$ 
                // Remove a cor da listas dos vizinhos
                 $C_j = C_j - color$ 
        }
         $\chi = 0$ 
        for each  $v_i \in V$  {
            if  $v_i.c > \chi$ 
                 $\chi = v_i.c$ 
        }
    return  $\chi$ 
}

```

Análise da complexidade

1. Note que a ordenação dos vértices pode ser realizada em $O(n \log n)$

2. A criação das listas de cores envolve trabalhar com n listas de tamanhos variáveis. Para o i -ésimo vértice, a lista de cores tem tamanho i , de modo que o número de operações é dado por:

$$1+2+3+4+\dots+n=\sum_{i=1}^n i$$

3. Mas é possível reduzir esse custo, sabendo que o número cromático em um grafo que não é o K_n é limitado superiormente pelo maior grau. Seja K o maior grau de um vértice em G , então basta que as listas de cores contenham K cores distintas e não n , ou seja, o número de operações gastas é:

$$1+2+3+\dots+K+K+K \leq nK$$

Portanto, a etapa de criação das listas de cores tem complexidade $O(nK)$

4. O loop principal do algoritmo percorre todo o vértice v de G uma vez e para cada vértice, acessamos os seus $d(v)$ vizinhos. Se utilizarmos um array estático, podemos apagar a cor color da lista C_i em $O(1)$. Logo, temos que a complexidade desse loop é:

$$d(v_1)+d(v_2)+\dots+d(v_n)=2m \text{ ,ou seja, } O(m) .$$

5. Por fim, o ultimo loop serve apenas para encontrar o maior rótulo utilizado como cor, o que é equivalente a encontrar o máximo do vetor. Assim, o custo é $O(n)$.

Portanto, o custo total é: $O(n \log n)+O(nK)+O(m)+O(n)$. Note que no pior caso, $m=O(n^2)$ o que nos leva a conclusão de que a complexidade do algoritmo Welsh and Powell é $O(n^2)$.

A seguir apresentaremos alguns exemplos de problemas que podem ser resolvidos através da coloração de vértices.

O Problema da Alocação de Frequências

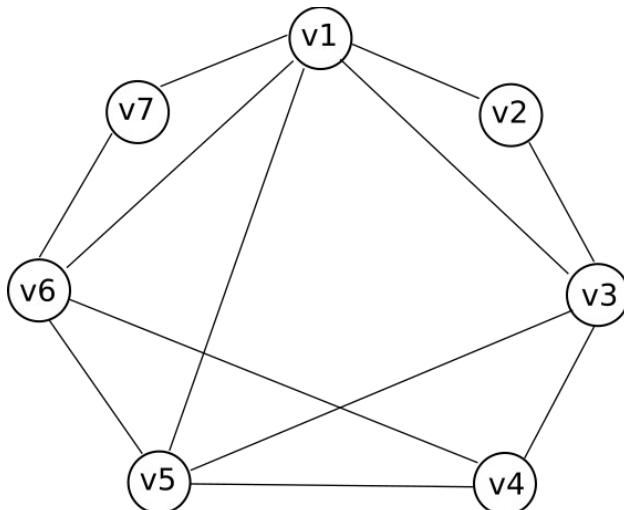
Considere 7 antenas de transmissão de sinais de telefonia. Sabe-se que devido a fatores como localização geográfica e tipo de serviço oferecido, as antenas possuem regiões de influência de modo que tem-se a seguinte matriz de interferências:

	v1	v2	v3	v4	v5	v6	v7
v1		x	x		x	x	x
v2	x			x			
v3	x	x		x	x		
v4			x		x	x	
v5	x		x	x		x	
v6	x			x	x		x
v7	x					x	

Essa matriz indica quando duas antenas interferem uma na outra com um x. Pergunta-se:

- a) Qual o menor número de frequências necessárias para que a comunicação se dê de forma correta?
- b) As antenas devem operar em qual configuração de frequência?

1º passo: Devemos construir o grafo de incompatibilidade. Neste grafo, dois vértices devem ser ligados por uma aresta se existe interferência entre as duas antenas.

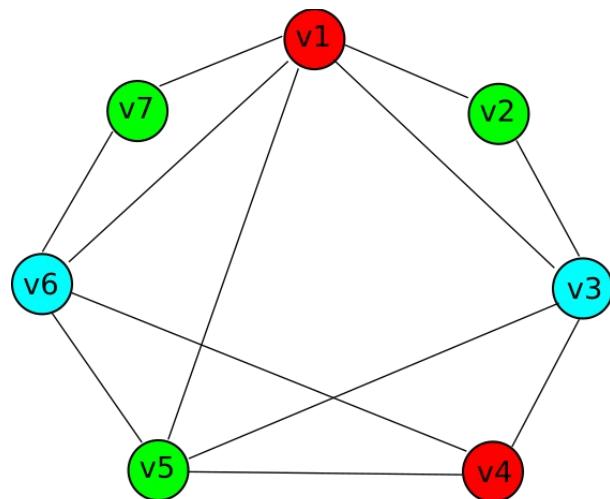


2º passo: Ordenar os vértices em ordem decrescente de graus: v1, v3, v5, v6, v4, v2, v7

3º passo: Definir as listas de cores para cada vértice

4º passo: Colorir os vértices iterativamente removendo as cores usadas das listas dos vizinhos.

	C1={1}	C3={1,2}	C5={1,2,3}	C6={1,2,3,4}	C4={1,2,3,4,5}	C2={1,2,3,4,5}	C7={1,2,3,4,5}
v1 ← 1	x	C3={2}	C5={2,3}	C6={2,3,4}	C4={1,2,3,4,5}	C2={2,3,4,5}	C7={2,3,4,5}
v3 ← 2		x	C5={3}	C6={2,3,4}	C4={1,3,4,5}	C2={3,4,5}	C7={2,3,4,5}
v5 ← 3			x	C6={2,4}	C4={1,4,5}	C2={3,4,5}	C7={2,3,4,5}
v6 ← 2				x	C4={1,4,5}	C2={3,4,5}	C7={3,4,5}
v4 ← 1					x	C2={3,4,5}	C7={3,4,5}
v2 ← 3						x	C7={3,4,5}
v7 ← 3							x



$$X(G) = 3$$

As partições obtidas são:

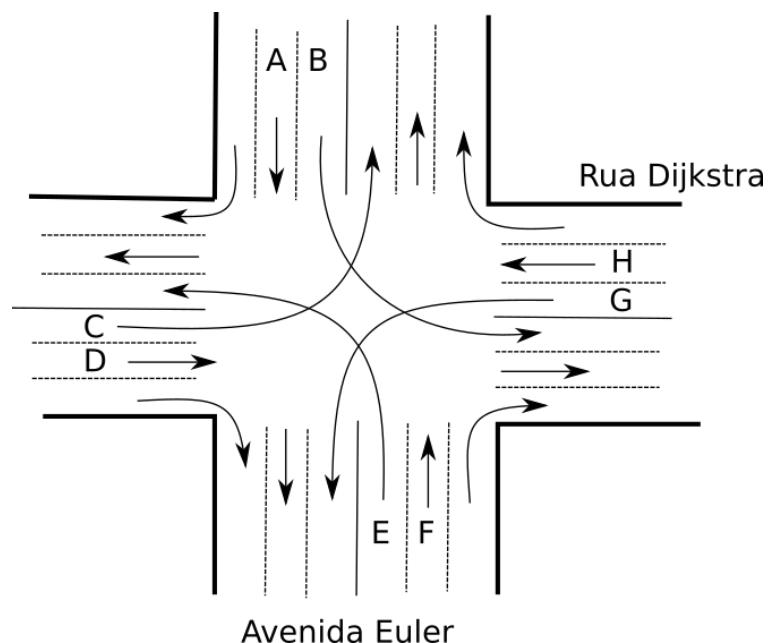
$$P_1 = \{v_1, v_4\}$$

$$P_2 = \{v_3, v_6\}$$

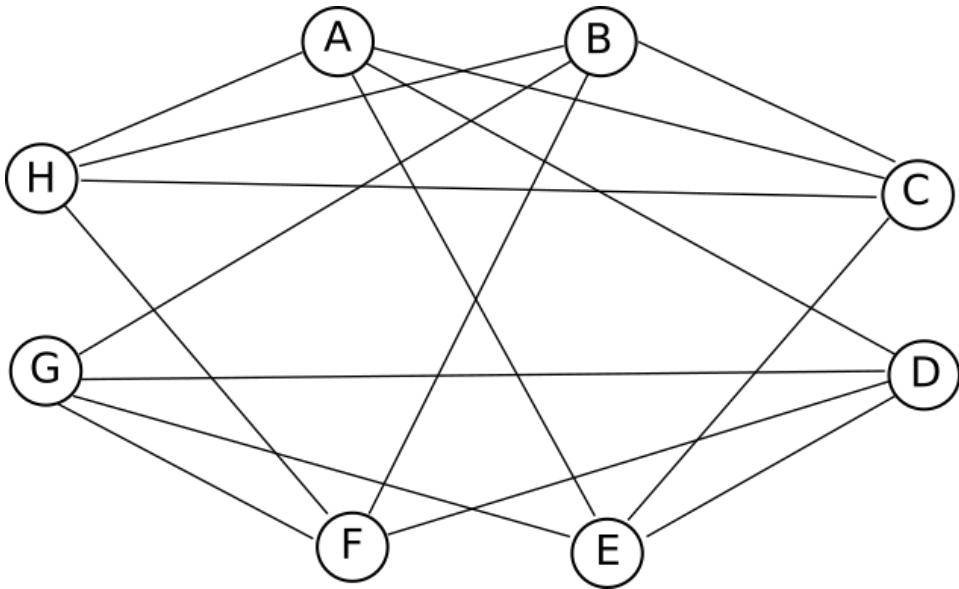
$$P_3 = \{v_2, v_5, v_7\}$$

O problema do semáforo

Dado o cruzamento de ruas a seguir, desenvolver um semáforo com o menor número de tempos possíveis. Qual deve ser o modo de funcionamento do mesmo.



1º passo: Consiste em gerar o grafo de incompatibilidade. Para isso cada uma das faixas de A a H deve ser um vértice e a cada possível cruzamento de faixas deve existir uma aresta (ou seja, deve existir uma aresta sempre que for possível ocorrer uma batida). O grafo resultante é ilustrado pela figura a seguir.



2º passo: Consiste na aplicação do algoritmo Welsh & Powell para colorir o grafo acima utilizando o menor número de cores possíveis. Como o grau máximo é 4 e o grafo não é completo, temos garantia de que o número cromático é menor ou igual a 4. A execução desse passo é deixada como exercício para o leitor.

O problema do aeroporto

Uma nova empresa aérea irá começar a operar com 7 aeronaves seguindo a programação de vôos (de A a G) definida pela tabela abaixo, sendo que todos os vôos partem de São Paulo e visitam cada uma das cidades listadas nas rotas na sequência em que elas aparecem:

Vôo	Rota
A	Florianópolis – Rio de Janeiro – Natal – Fortaleza
B	Curitiba - Campinas – Ribeirão Preto – Fortaleza
C	Belo Horizonte – Natal – Fortaleza – Manaus
D	Belo Horizonte – São José do Rio Preto – Rio de Janeiro
E	Belo Horizonte – Recife – Natal
F	Brasília – Ribeirão Preto – Fortaleza
G	Brasília - Presidente Prudente – Campinas

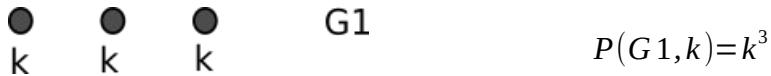
Devido ao número limitado de aeronaves, o diretor da companhia não quer mais de um vôo por dia visitando uma determinada cidade, ou seja, se dois vôos passam pela cidade X eles devem obrigatoriamente não estar alocados para o mesmo dia. Sendo assim, modelando o problema com um grafo, e utilizando coloração de vértices, determine o número mínimo de dias necessários para que a empresa opere de acordo com a sua política de funcionamento.

Uma pergunta natural que surge nesse momento é: de quantas maneiras podemos colorir um grafo G utilizando para isso exatamente k cores? O polinômio cromático é uma ferramenta matemática da teoria dos grafos algébrica que nos permite contar o número de k -colorações válidas de um grafo arbitrário G .

Polinômio cromático

Def: Para um grafo $G = (V, E)$, seja $P(G, k)$ o número de k -colorações válidas de G . Pode-se mostrar que $P(G, k)$ é um polinômio para todo grafo G .

Iremos iniciar com grafos simples e calcular os polinômios cromáticos fundamentais. Primeiro, iremos iniciar com um grafo nulo de 3 vértices. Note que nesse caso, podemos utilizar k cores em cada vértice, pois como não há arestas, nenhum vértice é vizinho de outro.



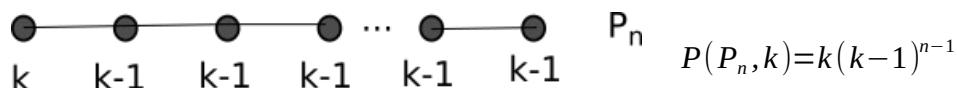
Note que ao adicionarmos uma aresta no grafo anterior, temos:



pois a cor do segundo vértice não pode ser utilizada para colorir o terceiro. Ao adicionarmos mais uma aresta, o polinômio cromático fica:



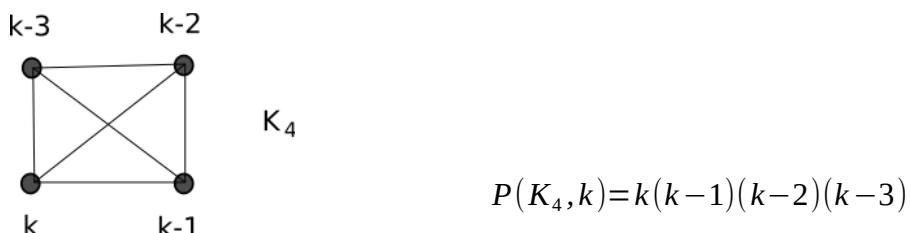
É fácil notar que o grafo caminho de n vértices, conhecido como P_n (path), possui o seguinte polinômio cromático:



O grafo completo K_3 (triângulo) possui outro polinômio fundamental:



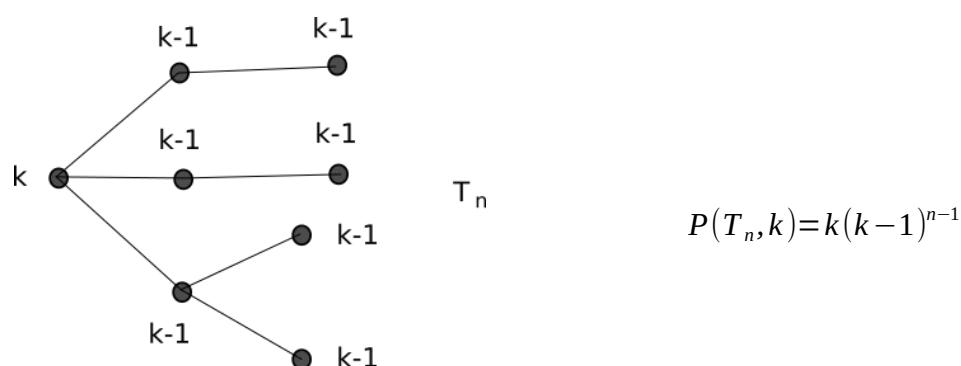
No caso do grafo completo de 4 vértices, K_4 , possui o seguinte polinômio cromático:



Generalizando para o grafo completo de n vértices:

$$K_n \quad P(K_n, k) = k(k-1)(k-2)\dots(k-(n-1))$$

No caso de árvores, note que o polinômio cromático é idêntico ao do grafo caminho. Aliás, isso já era esperado, uma vez que o grafo caminho é uma árvore.



Porém, uma pergunta que surge é: como calcular o polinômio cromático para um grafo arbitrário? O resultado a seguir é fundamental para responder a essa pergunta.

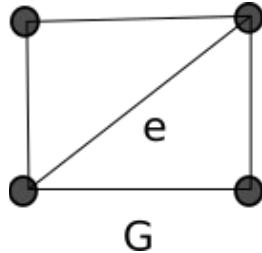
O Teorema fundamental da redução

Seja $G = (V, E)$ um grafo básico simples e e uma aresta do conjunto E . Então:

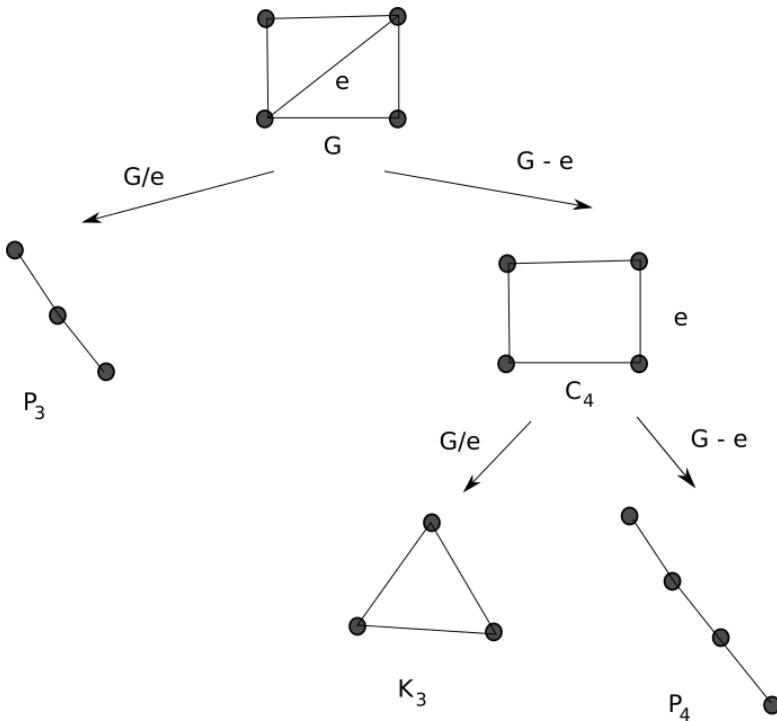
$$P(G, k) = P(G - e, k) - P(G/e, k)$$

onde G/e denota o grafo obtido a partir da contração da aresta e . Devido a complexidade matemática, iremos omitir a prova desse teorema. A seguir vamos aplicá-lo em um exemplo prático.

Ex: Determine o polinômio cromático de G a seguir:



Iremos aplicar o teorema fundamental da redução para gerar a árvore de decomposição do grafo. No primeiro estágio, devemos decompor os níveis da árvore até chegar em um polinômio fundamental e em seguida, voltar na árvore calculando os polinômios cromáticos. A árvore de decomposição é:



Iniciamos pela aplicação do teorema fundamental da redução em G :

$$P(G, k) = P(G - e, k) - P(G/e, k) = P(C_4, k) - P(P_3, k)$$

Como P_3 tem um polinômio fundamental, é folha da árvore de decomposição. Assim, temos:

$$P(P_3, k) = k(k-1)^2$$

Esta folha está encerrada. Vamos agora aplicar o teorema da redução em C_4 :

$$P(C_4, k) = P(C_4 - e, k) - P(C_4/e, k) = P(P_4, k) - P(K_3, k)$$

Note que ambos P_4 e K_3 possuem polinômios fundamentais, o que nos leva a:

$$P(P_4, k) = k(k-1)^3$$

$$P(K_3, k) = k(k-1)(k-2)$$

Como ambos são folhas, podemos voltar na árvore e computar o polinômio de C_4 como:

$$P(C_4, k) = k(k-1)^3 - k(k-1)(k-2) = k(k-1)[(k-1)^2 - (k-2)]$$

Expandindo o quadrado, temos:

$$P(C_4, k) = k(k-1)[k^2 - 2k + 1 - k + 2] = k(k-1)(k^2 - 3k + 3)$$

Agora que temos o polinômio de C_4 , podemos voltar e calcular o polinômio final de G como:

$$P(G, k) = k(k-1)(k^2 - 3k + 3) - k(k-1)^2$$

Colocando o termo $k(k-1)$ em evidência:

$$P(G, k) = k(k-1)[(k^2 - 3k + 3) - (k-1)] = k(k-1)(k^2 - 4k + 4)$$

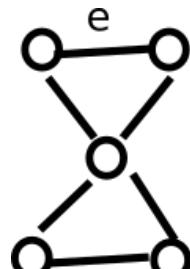
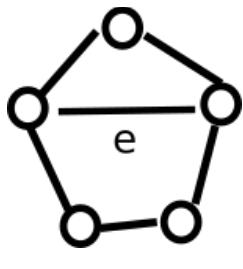
Fatorando o polinômio quadrático, finalmente chegamos em:

$$P(G, k) = k(k-1)(k-2)^2$$

Pergunta: quantas 5 colorações válidas existem para o grafo G em questão?

$$P(G, 5) = 5 \times 4 \times 3^2 = 20 \times 9 = 180$$

Exercícios: Utilizando o teorema fundamental da redução, encontre o polinômio cromático dos grafos a seguir. Mostre a árvore de decomposição.



"Solutions are not found by pointing fingers; they are reached by extending hands."
-- Aysha Taryam

Bibliografia

- CORMEN, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 4^a edição, The MIT Press, 2022.
- R. SEDGEWICK, K. WAYNE. Algorithms, 4th. ed., Addison-Wesley, 2011.
- LEVITIN, A. Introduction to the Design and Analysis of Algorithms, 3^a edição, Pearson, 2012.
- SKIENA, S. S. The Algorithm Design Manual, 2^a edição, Springer, 2008.
- KLEINBERG, J.; TARDOS, E. Algorithmic Design, Addison Wesley, 2005.
- S. DASGUPTA, C.H. PAPADIMITRIOU, U.V. VAZIRANI. Algorithms, McGraw-Hill, 2007.
- SZWARCFITER, J. L. Teoria Computacional de Grafos: Os algoritmos, Elsevier, 2018.
- ZIVIANI N. Projeto de algoritmos: com implementações em Java e C++. 2. ed. São Paulo: Cengage Learning, 2011.
- NICOLETTI, M. C.; HRUSCHKA, E. R. Fundamentos da Teoria dos Grafos para Computação, 2^a ed., Série Apontamentos, EdUFSCar, 2009.
- CLARK, J., DEREK, A. H. A First Look at Graph Theory, World Scientific, 1998.
- Geeks for Geeks – A computer science portal for geeks. <https://www.geeksforgeeks.org/>

Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor adjunto no Departamento de Computação da Universidade Federal de São Carlos e seus interesses em pesquisa são: filtragem de ruído em imagens e aprendizado de métricas via redução de dimensionalidade para problemas de classificação de padrões.